

Design and Implementation of a new Inter-Process Communication Architecture for Log-based HIDS for 100 GbE Environments

Bachelor Thesis

by

Paul Raatschen



University of Potsdam
Institute for Computer Science
Operating Systems and Distributed Systems

Supervisors:
Prof. Dr. Bettina Schnor
M.Sc. Max Schrötter

Potsdam, April 13, 2023

Raatschen, Paul

raatschen@uni-potsdam.de

Design and Implementation of a new Inter-Process Communication Architecture for Log-based
HIDS for 100 GbE Environments

Bachelor Thesis, Institute for Computer Science
University of Potsdam, April 2023

Thanks

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, April 13, 2023

Paul Raatschen

Abstract

Deutsche Zusammenfassung

Deutsche Zusammenfassung

Contents

1	Introduction	1
2	Background	2
2.1	Host-based Intrusion Detection / Prevention	2
2.1.1	Fail2ban	2
2.2	Inter-Process Communication	4
2.2.1	Types of IPC	4
2.2.2	IPC based logging	5
2.3	External Software	6
2.3.1	Hyperscan	6
2.3.2	io_uring	7
2.3.3	Trex	7
3	Design & Implementation	8
3.1	Requirements	8
3.2	Abstract Architecture	9
3.3	Choice of IPC Type	10
3.4	Shared Memory API	12
3.4.1	Design	12
3.4.2	Implementation	14
3.4.3	Write API	17
3.4.4	Read API	18
3.5	Proof-of-Concept IPS	19
3.6	Test Application	24
3.7	Other Applications	25
4	Evaluation	26
4.1	Test Environment	26
4.2	Experimental Design	27
4.3	Experiment 1: Fail2ban Replication	30
4.4	Simplefail2ban Measurements	32
4.4.1	Experiment 2: Logfile Variant	32
4.4.2	Experiment 3: Shared Memory Variant	35
4.4.3	Experiment 4: Shared Memory Features	39
5	Conclusion	44

List of Figures	45
List of Tables	46
List of Algorithms	47
A Abbreviations	48
B Source Files	49
C Measurements	50
Bibliography	62

1 Introduction

Ever since the advent of the public internet in the past century, bandwidths available to both private and commercial users have continuously expanded. While this opens up opportunities for new services and bandwidth-extensive applications, it also poses new security challenges for dealing with potentially malicious network traffic. In addition to traditional firewalls, Intrusion Detection Systems (IDSs) are a commonly used security measure to protect hosts in both public and private networks. They monitor system and network resources, to identify and respond to security breaches and potential attacks. Host-based Intrusion Detection Systems (HIDS) in particular, rely on information provided by the host system, one possible source for which are application logs. Traditionally, HIDS gather log information by parsing the logfiles maintained by applications or the Operating System (OS). One notable example for this, is the open source software Fail2ban. Fail2ban is an Intrusion Prevention System (IPS), that is widely used to protect servers against different network based attacks. It utilizes logfiles to identify and subsequently ban clients that fit predefined patterns of harmful behavior.

Previous work has shown, that the performance of Fail2ban scales poorly, when large amount of log messages need to be processed in a short time frame [1]. This constitutes a problem for high bandwidth networks, as millions of clients can send request simultaneously, leading to a significant influx of security relevant log events in case of an attack. If these messages are not processed, the system becomes vulnerable to security breaches. One possible reason for Fail2bans performance issues is the high latency associated with the file-based transmission of log messages. This inhibits its responsiveness in a high-bandwidth attack and consequently leads to Fail2ban being overwhelmed by the amount of log events to processes. Therefore, replacing log file parsing with faster methods of log message transmission, could improve the capability to handle the high load scenarios that can occur in modern networks.

The goal of this thesis will be the design and implementation of a new Inter-Process Communication (IPC) based architecture for the transmission of log messages, that is able to facilitate low latency communication between sender and receiver in the context of HIDS. Additionally, the design should be able to scale to multiple recipients, in order to accommodate more complex security system, in which several processes require access to a hosts application log. For the purpose of evaluation, a Proof of Concept (PoC) IPS will be developed, that utilizes the proposed IPC architecture to receive log messages and ban malicious clients in the style of Fail2ban. The PoC will be evaluated and compared to Fail2ban, to asses the viability of the new architecture.

This thesis will be structured as follows: The background section provides information on relevant concepts and introduces the problem setting in further detail. Subsequently, design and implementation of the new IPC architecture and the proof of concept IPS are presented. The evaluation section provides the experimental design and the results for the evaluation of both Fail2ban and the PoC. The final section concludes and offers an outlook for further development.

2 Background

The following section introduces the concept of HIDS with the specific example of Fail2ban and presents the problem setting this thesis aims to solve. In addition to that, a short overview over common types of inter process communication and existing IPC based logging solution is given. Finally, external libraries and other software used for the implementation and evaluation of the proof of concept IPS are introduced.

2.1 Host-based Intrusion Detection / Prevention

Intrusion Detection Systems are tasked with monitoring and collecting data from target systems, which is then further processed and analyzed, to identify potential threads and facilitate a response [2]. The idea of specialized software for detecting intrusion attempts and other security threads goes as far back as 1980, when James Anderson published a study on “Computer security threat monitoring and surveillance”[3]. It defined a threat to a computer systems as the: “potential possibility of a deliberate unauthorized attempt to access or manipulate information or render a system unreliable or unusable”[3, p.6]. Further, he suggested the use of automated tools to assist with security monitoring. In 1987, Dorothy Denning presented a seminal model for Intrusion Detection Systems, that proposed the use of pattern matching based on statistical analysis of audit records generated by a system, in order to detect abnormal user behavior [4]. Intrusion Detection Systems in general, can collect data from a multitude of sources. This allows for the distinction between Network-based Intrusion Detection Systems (NIDS) and Host-based Intrusion Detection Systems. NIDS monitor network interfaces and analyze captured traffic, while HIDS gather information directly provided by the hosts under their supervision. For the latter, this includes event logs of applications, as well as OS based information, such as user logins, file system operations or systemcalls. For analysis of the accumulated data, there are two commonly deployed strategies: 1. Misuse based detection relies on predefined patterns of misuse or malicious behavior, which are then matched against the observed data. 2. Anomaly based detection uses statistical analysis, to identify significant deviations from normally observed behavior, which, in principle, enables the detection of attack patterns, that have not been previously observed. [2]

2.1.1 Fail2ban

Fail2ban is an open source Intrusion Prevention System (IPS), that is widely used to protect hosts against a range of network-based attacks, such as for instance, brute-force login attempts[5]. Intrusion prevention system constitute a special class of IDS, that not only detects an intrusion attempt, but also initiates an active response, with the aim of preventing or mitigating the attack. To identify potentially malicious clients, Fail2ban uses a misuse detection approach based on application logs. For configured applications, Fail2ban actively monitors their log and parses new entries based on a predefined filter. Fail2ban uses configuration units called ‘jails’ , that allow for

the customization to different applications. A jail defines a path to an application log, the filter being applied to the log messages within the logfile and an action, that is executed on client matching the filter criteria. In addition to that, jails contain further parameters, such as the threshold of matches per client for the action to be executed, as well as the duration of the action. The filter component of a jail defines a set of regular expressions, that are used to identify certain events in a log, for example an unsuccessful login attempts or the exceeding of a request rate limit. The filter also obtains a clients IP address to identify the client and datetime information to determine, if the event occurred within a relevant time frame. Most commonly, the action issued by Fail2ban for eligible clients, is to ban the clients IP address. Fail2ban facilitates this via an `iptables` entry. `textttIptables` is a utility program for Linux systems, that allows the interaction with the `netfilter` kernel framework to implement packet filtering rules [6, 7]. When banning a client, Fail2ban add a `iptables` rule for the clients IP address that leads to the dropping of all subsequent network packets from that address, for as long as the rule is active.

Fail2bans netfilter-based approach of packet filtering has the disadvantage of scaling poorly for large traffic rates, since packets need to traverse several processing steps of the kernel network stack before they are ultimately discarded. In his master thesis, Florian Mikolajczak proposed the alternative usage of eBPF programs, as a more efficient way of implementing packet filtering for intrusion prevention systems[1]. The extended Berkley Packet Filter extended Berkeley Packet Filter (eBPF) is an interface of the Linux kernel, that allows the event based execution of verified, user defined programs within the kernel. Packet filtering through eBPF can be facilitated with an eBPF program, that is executed on incoming packets and delivers a verdict, of wether the packets should be dropped or further processed by the Kernel. Via the eXpress Data Path (XDP), eBPF programs can be attached to different hooks in the packet processing pipeline. For supported devices, programs can be attached in `XDP_DRIVER` mode, where they are executed as part of the network device diver routine, very early into the packet processing. Florian Mikolajczak adapted an existing eBPF program by Jesper Brouer and Andy Gospodarek to handle Internet Protocol Version 6 (IPv6) traffic as well and integrated it with Fail2ban to replace netfilter-based packet filtering. While he was able to demonstrate significant performance improvements over netfilter-based filtering, his measurements revealed, that Fail2ban has significant performance issues, when faced with a large influx of log messages.

Figure ?? shows the results of the Fail2ban measurement conducted by Florian Mikolajczak. In the experiment, a BIND DNS Server received unwanted requests at a rate of 1 million Packets per Second (PPS) from 254 different clients, which resulted in corresponding entries in the servers rate-limit log. Fail2ban was configured to ban clients with rate limit violations for 300 seconds. Initially, the performance was as expected. However, after the end of the first ban cycle, Fail2ban failed to renew the ban for some clients, leading to a significant amount of unwanted traffic still reaching the application. Closer inspection of Fail2bans behavior indicated, that the performance issues are likely the result of slow logfile parsing. Fail2bans rate of processing log entries appeared to be exceeded by rate of new entries, leading to Fail2ban falling increasingly further behind in the processing of logged events. This constitutes a problem, as it essentially makes Fail2ban and by extension the protected host, vulnerable to **DOS!** (**DOS!**) attacks. This problem setting provides the primary motivation for this thesis and is aimed to be resolved with the new IPC architecture that will be developed.

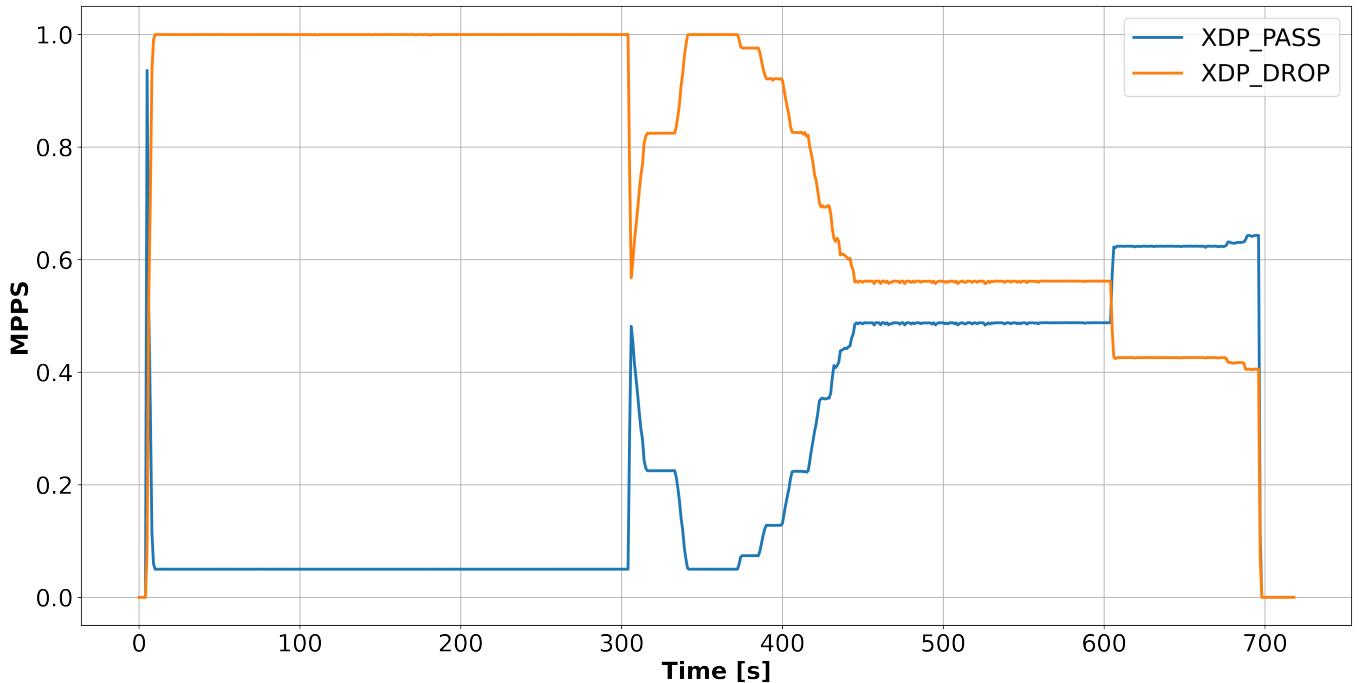


Figure 2.1: Results of experiment 1 from the master thesis of Florian Mikolajczak[1]. Fail2Ban, even in conjunction with more efficient eBPF filtering, performs poorly for large traffic rates.

2.2 Inter-Process Communication

2.2.1 Types of IPC

Inter-process communication allows the exchange of data between different processes through APIs provided by the operating system. Since the development environment for this thesis will be Linux, the focus is restricted to IPC APIs that are available on UNIX-like systems.

Shared Memory

A commonly used mechanism for sharing large amount of data between two processes on the same system is shared memory [8, p.301ff.]. Shared memory allows the allocation of a memory segment in RAM, that can be mapped into address space of several processes. The memory segment has associated file associated permissions, allowing the creating process to control the read and write access of other processes. Linux provides two main ways of creating shared memory in the System V shared memory API [9] and the newer POSIX shared memory API [10]. Processes can essentially treat shared memory like any other valid memory in their address space, which allows for fast and flexible application. A disadvantage of shared memory, at least for the aforementioned APIs, is its restriction to the local system boundaries, though solutions for network based remote direct memory access (RDMA) exist[11].

Named Pipes / FIFOs

Pipes, specifically named pipes, are a way of facilitating unidirectional data transfer between processes on a UNIX system. Named pipes have specific read and write file descriptors and are

associated with a special file in the filesystem, which can be accessed by different processes. Pipe can be written to and read from via the write and read system calls. The data is transmitted in a First in First out (FIFO) manner, hence why named pipes are also commonly referred to as FIFOs. The maximum capacity up to which a pipe can buffer data is limited, which by default is 65535 bytes on linux, but can be adapted by a user. [12]

Sockets

Sockets are another common **IPC!** type for data transfer, that allow for one-to-one as well as one-to-many communication via a range of protocols[13, p.57ff.]. Sockets send data in formatted packets, that can be transmitted on a local host or via a network. For communication on a local system, UNIX systems offer UNIX Domain sockets. UNIX Domain sockets can be bound to a valid path in the filesystem, which serves as a way of addressing the socket from another processes [14]. For data transfer beyond the local system, the Internet protocol Internet Protocol (IP) in conjunction with the Transmission Control Protocol Transmission Control Protocol (TCP) or User Datagram Protocol User Datagram Protocol (UDP) is most common. TCP offers connection oriented data transmission, that requires an orderly connection establishment through a hand-shake between the communicating parties. It further ensures a reliable transfer of data, by detecting transmission errors through acknowledgment and retransmitting unacknowledged packets. UDP in contrast, provides no guarantees on reliable transfer, with the benefit of lower latency and less communication overhead. [13, p.29ff.]

Message Queues

Finally, message queues are another commonly used IPC mechanism that allow the exchange of messages between processes. Linux natively supports the System V [9] and POSIX [15] message queue APIs, but third party implementations exist as well, for instance the ZeroMQ library [16]. Message queues essentially function as a buffer, which can store messages up to a certain capacity. A writing process adds messages to the queue, while reading processes remove messages from the queue. Unlike with sockets, writing and reading can occur asynchronously i.e. messages do not have to be read immediately after being added to a queue. ZeroMQ specifically supports a range of communication patterns, among them the publish-subscribe pattern, where a writer sends messages to multiple subscribed readers or the request-reply pattern, that can be used to implement remote procedure calls.

2.2.2 IPC based logging

Other than traditional file-based logging, syslog is a commonly used protocol to facilitate network based-logging on UNIX-based systems. The syslog protocol was first developed in the 1980s by Eric Allman, as a solution for centralized logging and ultimately standardized by RFC 5424[17]. Syslog operates on a server-client basis, where logging clients send messages to a syslog server, that can be located on the host or a remote machine. This allows the aggregation of log events from different sources and also serves to functionally separate the generation of log messages from storing, analyzing and further processing. Syslog messages have a standardized format, that contains sender information, date and time of the event, as well as a facility and severity code. The facility indicates the category of the logging application, while the severity indicates

2 Background

urgency and impact of the logged event¹. The syslog protocol is most commonly used with UDP but does not specify a transport protocol and can also be used in encrypted form, via the Transport Layer Security (TLS) protocol. Rsyslog is a modern implementation and extension of the syslog protocol developed by Rainer Gebhard[18]. It is intended as a high-performance log processing system for enterprise systems, that allows the aggregation of log events from a variety of sources such as files, named pipes, sockets or message queues via the Advanced Message Queue Protocol (AMQP)[19].

Logstash is another commonly used system, that supports IPC based data aggregation [20]. It provides input plugins for a range of inputs including files, sockets, pipes and messages queues. Logstash is part of the Elasticsearch technology stack, that allows distributed aggregation, processing and analysis of various data sources. Log aggregators like Rsyslog and Logstash have application in a security context, as they can be integrated as part of Security Information and Event Management (SIEM) systems. SIEM systems collect data from various security relevant contexts, such as firewalls or intrusion detection systems, to provide comprehensive real-time security monitoring and analysis of complex systems[21].

Regarding other high performance logging systems, the literature is, to the best of my knowledge, rather slim. Jeong et al. present a shared memory based logging system for embedded UNIX-based applications [22]. They use a memory mapped file to buffer log messages between the logging application and a daemon processes for forwarding the messages. Their logging systems achieves a 10 times larger message throughput and 100 times lower latency than traditional syslog, which they attribute to the pure use of user-level IPC that avoids costly context switches to the kernel.

2.3 External Software

2.3.1 Hyperscan

Hyperscan is a open source regular expressions matching engine developed by Intel [23]. It is specifically designed for high performance use cases, such as the application in security contexts and is being used by the intrusion detection systems Snort and Suricata. The process of regular expressions matching with Hyperscan is separated into compile- and run-time. At compile-time, a set regular expressions in string representation are compiled into a database, with additional configuration options. These include the processing mode, which can be streaming, block or vectored mode. Streaming mode allows the scanning of a continuous data stream, which is facilitated via a state maintained between function calls. Block and vectored mode operate on data that is readily available and can be scanned in a single call. Other compile options include case less and multiline scanning as well as instruction Hyperscan to only match an expression once or to report the leftmost start index of a match². The latter comes with detrimental performance implications Hyperscan allows the compilation of multiple patterns into a database, which can be scanned for in parallel. For scanning hyperscan allows the user to define a callback function that is called when a match occurs. The matched pattern can be identified with an id parameter.

¹A full list of codes can be found in [17, p.10-11]

²By default, only the end index of a match is provided.

For the purpose of this thesis, Hyperscan will be used as a regular expression engine for the matching of patterns in log messages received by the proof of concept IPS.

2.3.2 io_uring

io_uring is an asynchronous Input / Output (IO) framework of the Linux kernel, that was first introduced in kernel version 5.1 [24]. In contrast to traditional read and write systemcalls, io_uring provides a non-block writing to or reading to file descriptors. This is facilitated via an in memory queues, that are shared between a user process and the kernel. The first queue is a submission queue, that the application uses to register IO operations to the kernel. The second queue is a completion queue, that the kernel fills with information on completed IO requests. An application can register several IO operations, which are ultimately submitted to the kernel. The application can then continuous execution, while the IO operation is asynchronously executed by the kernel. Via the completion queue, the application can poll the status of the submitted operations and receive information about their completion.

Liburing is a library written by Jens Axboe, that provides a high level interface to the io_uring framework [25]. It allows the setup of a a io_uring instance, which can be subsequently used to conduct IO operations on files but also other types of file descriptors such as sockets.

For the purpose of this thesis, liburing will be used to implement asynchronous file reading and writing for both the proof concept IPS as well as auxiliary applications.

2.3.3 T Rex

TRex is a traffic generating tool developed by Cisco System [26]. It supports the generation of both stateless as well stateful traffic, from network up to application layer protocols. TRex is build with the Data Plane Development Kit framework, that offers user space based packet acceleration, enabling TRex to reach traffic rates up to 200Gbs, given appropriate hardware support. Traffic can be generated either via pcap files or Python scripts, using the packet manipulation library scapy [27].

For the purpose of this thesis, TRex will be used to conducted measurements and evaluate the proof of concept IPS. During initial measurements, its was discovered, that TRex struggles to maintain the advertised traffic rates for stateful traffic in certain scenarios. More specifically, the absence of an acknowledgement packet to a TCP-syn packet sent by TRex appears to inhibit performance. Hence, measuring a Denial of Service (DoS) scenario as described in section 2.1.1 for TCP based traffic is problematic, as syn packet for banned clients emulated by TRex will not be answerer by server protected by Fail2ban or the proof of concept IPS. For this reason, all measurements conducted in this thesis will be UDP based.

3 Design & Implementation

The following section introduces the design and implementation of the proposed IPC architecture and the proof of concept IPS, as well as auxiliary libraries and applications. First, the requirements and overall purpose of the design are defined and translated into an abstract architecture. Subsequently, the implementation for a concrete IPC type is presented.

3.1 Requirements

The primary goal of the new IPC based logging architecture is the ability to handle high load scenarios, like the DoS scenario discussed in section 2.1.1. This includes the basic functional requirement of offering an API that can be used to send data to and receive data from different processes. Beyond that, the architecture should be usable in a realistic context. For this purpose, the following requirements should be satisfied, in order of importance.

- **Low latency**

As discussed in 2.1.1, the primary issue of file based logging in the context of IPS appears to be a lack of transmission speed. Hence, offering low latency data transfer is a critical aspect of the new architecture. In a DoS scenario, the IPS need to receive information about malicious clients as fast as possible, in order to take countermeasures that lower the load on the system.

- **Low overhead**

Another problem with file based logging is, that both read and write operations require systemcalls, which lead to costly context switches between application and kernel. Using traditional file I/O, read and write operations are blocking, meaning that a program can only resume, once the operation has concluded. In a high load scenario, where potentially millions of log messages are written per second, this can cause significant communication overhead for both reader and writer. It also looses critical execution time to waiting, which, on the side of the writing server, lowers the availability of the service and on the reading IPS side, reduces responsiveness to the ongoing attack. The new architecture should therefore offer low overhead communication, ensuring that both writer and reader are not spending a majority of their execution time writing or receiving log messages.

- **High bandwidth**

In conjunction with low latency, high bandwidth is required to ensure a fast transfer of large amounts of data. When processing millions of log events per second, the architecture needs to be able to execute a single data transfer quickly, but also allow for a large amount of data to be transmitted at once, to avoid a bottleneck between writer and reader.

- **Reliability**

While not the primary concern, the architecture should in principle be able to transfer log messages reliably, even under a high load. Hence, message loss or corruption should be kept to a minimum. This is to ensure, that the IPS or other reading application are aware of entirety of all logged events and no potentially security relevant event is missed.

- **Scalability**

An advantage of file based logging is, that logged events are stored persistently and also accessible to a multitude of applications. Log data generated by an application may be relevant in different contexts, such as intrusion prevention, SIEM or other non-security related uses. Therefore, the architecture should be able to scale to multiple readers that can access the log data in parallel.

- **Integrability / Portability**

Finally, to be of use in a realistic context, it is required that the architecture is able to integrate with existing applications. Since the goal of this thesis is a proof of concept and not an API ready for use in production, this will be a lower priority requirement. However, the design should at a minimum offer a well defined and easily usable API, that can be realistically integrated into a real application and potentially further developed beyond the scope of this thesis. As part of that, the design should be portable to different systems and ideally not rely on special hardware or software support, other than what is commonly present on most Linux systems. The architecture should also be flexible i.e. impose as little restrictions as possible on the user, in order to be adaptable to a wide range of use case scenarios.

The architecture proposed on this thesis will likely not be able to satisfy all listed requirements to the fullest. Potential tradeoffs between requirements exist and will have to be considered in the following design. Where tradeoffs are impossible to avoid, the architecture should ideally offer configuration options, that allows the user to decide, which requirements should be prioritized.

3.2 Abstract Architecture

To further illustrate the purpose and requirements of the desired architecture, an abstract architecture is provided. The goal is to provide a high-level design, that formalizes a set of functional requirements, but leaves enough flexibility to be implemented with different IPC solutions.

Figure ?? presents the abstract design for the proposed IPC architecture. The primary assumed use case is a producer consumer scenario, in which a single application writes log messages that are read by k receiving processes. The transfer of data is assumed to be unidirectional, i.e. the roles of reader and writer are static for the duration of the communication. An exception to this can be auxiliary messages for the setup, teardown or reconfiguration of the communication, that may require readers to send data to the writer. The writer is assumed to be a server, that receives client requests over a network. On modern multi-core systems, web servers like nginx and apache or a DNS server such as BIND usually use multiple threads to handle incoming requests. This allows for the parallel distribution of work onto the available CPU cores, making more efficient use of the system's resources. To efficiently accommodate a multi-threaded writer, the

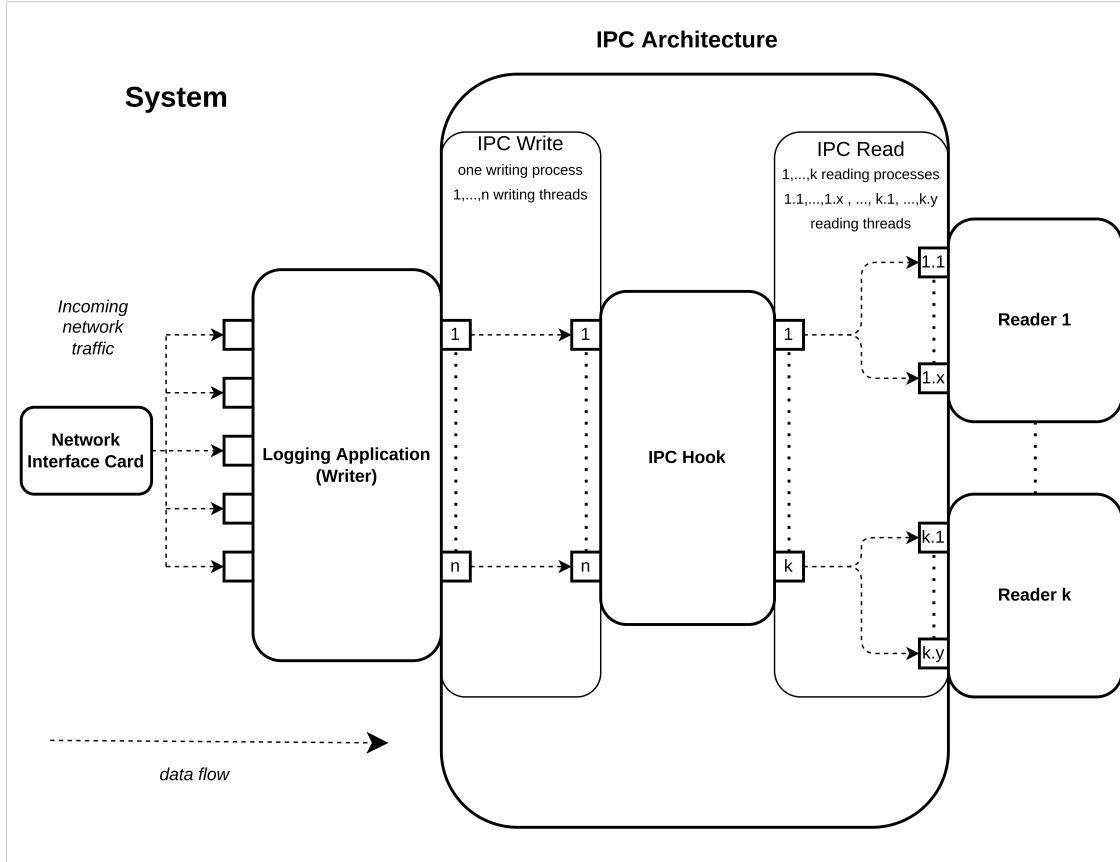


Figure 3.1: Abstract IPC architecture

IPC architecture is required to support a thread-safe write operation. Hence, it must be possible to write in parallel from n threads, without race conditions that could cause data corruption, or the need for additional thread synchronization on the writer side. On the reader side, the architecture must be able to support multiple reading processes, that each receive the entirety of the written messages and are able to read in parallel. To allow load balancing on the reader side as well, the read operation should also be capable of multi-threading. For the multi-threaded read within a reading process, the operation should enable a balanced and starvation-free distribution of messages to the calling threads.

Given the requirements and abstract architecture, the goal is to find a concrete implementation, that satisfied the proposed design.

3.3 Choice of IPC Type

During the development of this thesis, all of the **IPC!** mechanism introduced in 2.2.1 were considered as suitable candidates for the proposed architecture. The initial objectives was to create multiple implementations, that each use a different IPC type. This would have allowed to compare their performance empirically and make a more informed decision as to which IPC type

is most appropriate for the given requirements. Due to time issues, the idea was abandoned in favor of a single implementation.

Shared memory was the first candidate considered for the design. The main reason for this is speed, since low latency communication is the primary requirement for the proposed architecture. As aforementioned, shared memory has the significant advantage of not involving the kernel for read or write operations. As long as the overhead for synchronizing the communicating processes is kept low, shared memory essentially operates at the speed of normal memory accesses, which is unlikely to be surpassed by any kernel-based IPC method. Shared memory also grants great flexibility in the design of the API. This also provides a challenge, as, to the best of my knowledge, no standard for the shared memory based transmission of log messages exists, making it harder to integrate the architecture with real world applications. Also, synchronization for multi-threaded and multi-process use will have to be implemented as part of the API. Another downside to shared memory is, that it does not trivially scale beyond the local system, which could be desired for more complex systems, where IPS and server do not share the same host.

Named pipes are another potential option to implement the design. Unlike shared memory, they already provide a well defined read and write API, that can be integrated into the proof of concept IPS as well as a log aggregator such as Rsyslog or Logstash. Additionally, pipes are safe for multi-process and multi-threaded reading. Multi-threaded writing by a single process is also thread-safe, as long as the data written is smaller than `PIPE_BUF` (4096 bytes on Linux)[12]. A downside to the usage of pipes is the requirement of systemcalls for IO operations. Especially, since thread-safe writing only allows for small message sizes, making it more complicated to reduce the amount of systemcalls through batched write operations.

A socket based implementation was also considered. A straightforward strategy for implementing the one to many communication required by the abstract architecture, would be the use of IPv6 multicast. IPv6 multicast allows several processes to join a distinct multi-cast group. Each member of the group receives the packets sent to the multicast address. This would additionally allow support of the syslog protocol, which could make for easier integration of the architecture with existing logging applications. The architecture could also scale beyond the local system to support network-based logging. A downside of a socket based approach, same as for named pipes, is the overhead associated with involving the kernel.

Lastly, a message queue design, using the ZeroMQ library was considered. ZeroMQ is a logical choice, since the library natively supports a publish subscribe pattern, that could be used to implement the required one to many communication. In addition to that, Rsyslog and Logstash support ZeroMQ sockets as input, which would allow easy integration with the architecture. One issue with a ZeroMQ based implementation is, that the `SO_REUSEPORT` flag, for reusing a single socket among several threads, is not supported by ZeroMQ[28]. ZeroMQ sockets are not otherwise thread-safe¹[16]. Hence, to the best of my knowledge, a publish subscribe pattern with a multi threaded producer can not be trivially implemented without external synchronization.

Ultimately, the decision was made, to develop a shared memory based implementation of the architecture. The main motivation for this is, that it is the most promising candidate for low-latency, high bandwidth communication, which is a critical aspect of the design. However, the proof of concept IPS will be designed as modular as possible, so that implementations based on other IPC types can be integrated for future development.

¹<https://zguide.zeromq.org/docs/chapter2/>

3.4 Shared Memory API

The following section presents the design and implementation of the new shared memory based architecture for the transmission of log messages.

3.4.1 Design

For the base structure of the shared memory design, a ring buffer was chosen. A ring buffer is a queue-like data structure, used to store multiple entries in a sequence. The entries are processed FIFO manner, hence data is being read in the same order that is was written. Ring buffers are a commonly used structure to buffer data streams, especially for network applications [29]. They are usually realized as a fixed size array with additional pointers, that indicate the current location of readers and writers within the buffer. When the end of the buffer is reached during a read or write operation, the pointer wraps around to the begin of the buffer, threatening the array, as if start and end were connected in a circular shape.

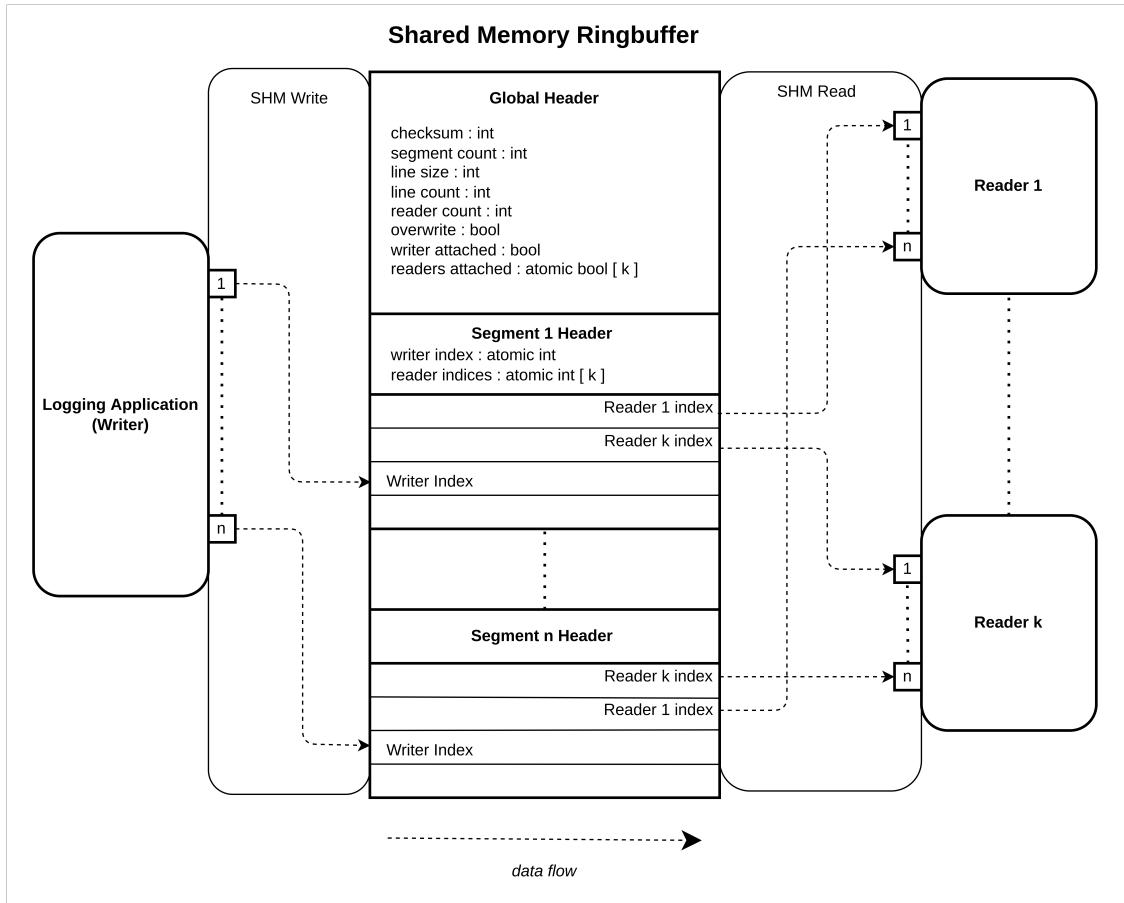


Figure 3.2: Architecture for the single-writer, multi-reader shared memory ring buffer for the transmission of log messages.

Figure ?? display the proposed ring buffer architecture, adapted from the abstract design in Figure ???. The shared memory region consists of a global header as well as a variable amount of segments, that each have their own segments header. Each segment constitutes a ring buffer that is independent of the other segments. The global header contains information that is global to the entire buffer and is used for initialization, when processes attach to the buffer. The buffer parameters within the global header are initialized by the processes creating the buffer and are supposed to be constant for the lifetime of the shared memory segment. The segment count variable indicates the amount of segments within the buffer. As illustrated in ??, the number of segments is supposed to map 1:1 to the number of writing threads. Since each writing threads has their own ring buffer, no synchronization for access to the write pointers is required. The individual entries within the segment ring buffers are fixed size lines, the length of which in bytes is determined by the line size field in the global header. Each line corresponds to one log message, the same way a line in a logfile would. Limiting buffer entries to a fixed maximum size has some obvious disadvantages. The primary concern is inefficient use of memory, as the line size will have to be chosen to accommodate the largest possible log message, since multi line messages are not intended in the design. If log messages vary greatly in length, this may cause a lot of unused memory for shorter messages. The fixed size was chosen for a more convenient implementation of the overwrite feature, which is enable with the overwrite filed in the global header and will be covered in more detail later. Allowing for variable length entries is still an important consideration for further development of the design. The line count variable in the global header determines the number of entries each segment ring buffer has. Making all segments equal in size implicitly assumes, that the distribution of log messages written is equal among the writing threads. However, variable size segments could be trivially added to the design, should a use case for that emerge. Finally, the global header contains a field for the number of allowed readers, as well as boolean fields, that indicate the attachment of the writer and the readers. Keeping the number of allowed readers fixed limits the ability of dynamically adding new readers, but makes the implementation more convenient. The attachment field are used for reference counting, of how many processes are attached to the buffer. When a reader attaches to the buffer, it will iterate through the readers attached array in the global header, in search of the first entry that is not set to true. If an entry is found, the reader will set the value of the entry to true and the index of the entry becomes the readers reader id. Since writing to the attachment fields is implemented via atomic compare exchange operations, no race conditions exist for simultaneously attaching readers. Similarly, the writer uses the write attached field when attaching to the buffer.

Each segment of the buffer begins with its own header, that consist of a write index and an array of read indices, that is reader count elements long. The index corresponds to a line in the segment and is used to keep track of the current position of the reader or writer in the buffer, as well as to synchronize reader and writer. Both reader and write indices are implemented as atomic variables. Hence, read and write operation are executed like a single CPU instruction. This eliminates the possibility, that a processes can read the index of another processes while it is being written and potentially obtain a corrupted value. Using atomic indices allows for lock free synchronization of reader and writer, which avoids the performance impact of having to switch threads between waiting and execution states that result from lock-based synchronization. However, atomic operations are also more costly compared to regular reads writes ². The lines

²The performance evaluation of atomic vs lock based synchronization is not part of this thesis, but could be a

of the segment begin below the segment header. The segment buffer capacity in bytes is given by line size "*" line count and the capacity of the entire buffer is given by segment capacity "*" segment count.

3.4.2 Implementation

In the following, the implementation of the design in figure ?? and relevant API functions is presented. The entire API was implemented as a static library in C. The associated header and source files are `shm_rbuf.h` and `shm_rbuf.c`, which can be found under `src/lib` in the Git repository for this thesis [30]. All API functions return numeric error codes, the meaning of which is also documented in `shm_rbuf.h`.

```
1 // Creates the ringbuffer or attaches to an existing one
2 int shmrbuf_init(union shmrbuf_arg_t * args,
3                   enum shmrbuf_role_t role);
4
5 // Detaches from the ringbuffer and removes the memory
6 // segment, if no other process is attached
7 int shmrbuf_finalize(union shmrbuf_arg_t *,
8                      enum shmrbuf_role_t role);
```

Algorithm 3.1: Initialization and cleanup function for the shared memory ring buffer.

Algorithm ?? displays the function signature for the initialization and finalizations functions. `shmrbuf_init` is used to create or attach to an existing shared memory buffer and has to be called before read or write operations on the buffer can be executed. `shmrbuf_finalize` is the corresponding function to detach or destroy the shared memory buffer and should be called upon ending the interaction with the buffer, for instance when exiting an application. `shmrbuf_finalize` will only destroy the shared memory segment, if no other process is attached. Otherwise the memory will simple be unmapped and the attachment field in the global header (as described in 3) is released. `shmrbuf_init` and `shmrbuf_finalize` can be called by both reader and writer processes, which is why a role has to be specified as an additional parameter in the function call. The role determines, what type of struct the union pointer `shmrbuf_arg_t` is being cast to, which contains further parameters used by the functions.

Algorithm ?? displays the struct, that is used by a writing processes, to store and pass related parameters to the API functions. In the current implementation, creation of a buffer can only be performed by a writing process. The reason for this is, that parameters such as the line size of segment count need to be tailored to the writing process. The writer creates a new buffer by parameterising a `shmrbuf_writer_arg_t` structure with the desired parameters for the buffer and calling `shmrbuf_init`. In addition to the already discussed parameters line size and line, segment and reader count, a key parameter is required. The key has to be a valid filepath within the file system and is used by System Vs shared memory API, to identify the shared segment³. If

consideration for a potential further development of this architecture.

³The System V shared memory API [9] was used instead of the more modern POSIX API, since its support for memory allocation with huge pages was more convenient. Allocating the buffer with huge pages, should make read

```

1 struct shmrbuf_writer_arg_t
2 {
3     const char * shm_key;
4     uint16_t line_size;
5     uint32_t line_count;
6     uint8_t segment_count, reader_count;
7     struct shmrbuf_global_hdr_t * global_hdr;
8     struct shmrbuf_seg_whdr_t * segment_hdrs;
9     int flags, shm_id;
10 };

```

Algorithm 3.2: Structure to store writer parameters for the shared memory ring buffer.

no shared memory segment referenced by the given path exists, a new segment of appropriate size will be allocated and mapped into the calling processes address space. Subsequently, its global header will be initialized with the parameters provided in the `shmrbuf_writer_arg_t` structure.

```

1 struct shmrbuf_global_hdr_t
2 {
3     uint32_t checksum;
4     uint8_t segment_count, reader_count;
5     uint16_t line_size;
6     uint32_t line_count;
7     bool overwrite;
8     atomic_bool writer_att, first_reader_att;
9 };

```

Algorithm 3.3: Structure to represent the global header of the shared memory ring buffer.

Algorithm ?? displays the structure for the global header of the shared memory buffer. When the header has been initialized the checksum field is set to the sum over all constant header fields of the header. This is done as an additional security check for other processes attaching to the buffer. If the checksum is not correct, `shmrbuf_init` will fail, to avoid writing to or reading from potentially corrupted memory. After initializing the header, `shmrbuf_init` sets the indices of all segment headers to zero. As an alternative to creating a new buffer, a writer can also reattach to an existing buffer. When reattaching, the buffer parameters provided in `shmrbuf_writer_arg_t` are ignored. To enable reattachment, the `REATT` flags has to be specified in the `flags` field of the `shmrbuf_writer_arg_t`, otherwise the call to `shmrbuf_init` will fail for an existing buffer. Reattachment is included in the API, so the writing application can be restarted (for instance for maintenance) without the need to destroy the shared memory segment, if readers are attached. The overwrite feature is also specified via the `flag` field by specifying the `OVWR` flag. Finally, `shmrbuf_init` initializes an array of segment header structures in `shmrbuf_writer_arg_t` and

and write operations on a large buffer more efficient, since fewer entries are required in the translation lookaside buffer (TLB), making TLB misses less likely.

3 Design & Implementation

sets the `global_hdr` pointer to point to the start of the shared memory segment.

```
1 struct shmrbuf_seg_whdr_t
2 {
3     atomic_uint_fast32_t * write_index, * first_reader;
4     void * data;
5 }
```

Algorithm 3.4: Structure to store writer information for a segment of the shared memory ring buffer.

Algorithm ?? displays the structure used in the segment header array. The structure does not represent the actual segment header the way its layed out in memory, but holds convenience pointers for a segment, that are used for writing operations.

For readers, attaching to an existing buffer is also done with a call to `shmrbuf_init`. This functions analogous the writer, with the exception that the shared memory segment pointed to by the provided path has to exist. Algorithm ?? displays the structure used for storing reader parameters.

```
1 // Reader parameters
2 struct shmrbuf_reader_arg_t
3 {
4     const char * shm_key;
5     int shm_id, flags;
6     uint8_t reader_id;
7     struct shmrbuf_global_hdr_t * global_hdr;
8     struct shmrbuf_seg_rhdr_t * segment_hdrs;
9 }
```

Algorithm 3.5: Structure to store reader parameters for the shared memory ring buffer.

If the attachment to the shared memory segment and the checksum were successful, the reader will obtain a unique reader id, as described in section 3. If all reader slots are already occupied, the `shmrbuf_init` call fails. Subsequently, `shmrbuf_init` will initialize the pointers to the global header and the array of segment headers structures in the provided `shmrbuf_reader_arg_t` structure. Readers have their own structure for storing segment header information, which is displayed in Algorithm ??.

```
1 struct shmrbuf_seg_rhdr_t
2 {
3     atomic_uint_fast32_t * write_index, * read_index;
4     pthread_mutex_t segment_lock;
5     void * data;
6 }
```

Algorithm 3.6: Structure to store reader information for a segment of the shared memory ring buffer.

The structure includes a pointer to the index of the reader within the segment header. Which read index of a segment header is assigned to a reader, is determined by their reader id. Additionally, the structure contains a mutex lock, that is used to synchronize access to the segment when reading with multiple threads. After a successful call to `shmrbuf_init`, both reader and writer can start using their respective APIs for the buffer.

3.4.3 Write API

```

1 // Writes a single line to a segment
2 int shmrbuf_write(struct shmrbuf_writer_arg_t * args,
3                     void * src, uint16_t wsize,
4                     uint8_t segment_id);
5
6 // Writes multiple lines to a segment
7 int shmrbuf_writev(struct shmrbuf_writer_arg_t * args,
8                     struct iovec * iovecs,
9                     uint16_t vsizes,
10                    uint8_t segment_id);

```

Algorithm 3.7: Write API for the shared memory ring buffer.

Algorithm ?? displays the write API for the shared memory ring buffer. `shmrbuf_write` writes a single log message to a specified segment within the buffer. The segment being written to has to be referenced with an id, which is simply its index within $\{0..n\}$, where n is the number of segments. The `src` argument is a pointer to the string that should be written to the buffer and `wsize` specifies the length of the string. If `wsize` exceeds the line size specified in the global header of the buffer, the write operation will fail with a size error. When issuing a call to `shmrbuf_write` without overwrite, the function will atomically load all reader indices in the segment header and check their distance to the write index. If no reader is directly ahead of the write index, the line will be written to the buffer and the writer index will be atomically updated to its new position, otherwise, the function call returns with an error. If the overwrite operation is specified in the header, synchronization between writer and reader is effectively disabled and the write operation will conclude, regardless of the readers positions. This incurs the risk, that a reader and writer simultaneously access the same line, which may potentially corrupt the data being read. An advantage of overwrite is, that the writer avoids costly atomic operations and also can't be blocked by a slow or stale reader. The performance of the overwrite feature will be evaluated in section 4. The write API provides a second write function in `shmrbuf_writev`. Instead of writing a single line, the function receives an array of `iovec` structures. The `iovec` structure defined in `struct_iovec.h` within the standard C library, is a generic container for vectored IO operations. It contains a void pointer to a data field and a variable to specify the length of the data in bytes. `shmrbuf_writev` works analogous to `shmrbuf_write`, with the only difference that `vsizes` lines referenced the corresponding `iovec` structure in `iovecs` will be written to the buffer. Performance inspection of `shmrbuf_write` with the benchmarking tool perf [31] revealed, that about 75% of the execution time is being spent for executing the atomic operations. Hence,

using a vectored write operation should improve performance, as the synchronization only has to be applied once for multiple log messages. For the sake of performance, neither write function implements synchronization for the write index. Calling a write operation on the same segment from two different threads, is therefore not thread-safe.

3.4.4 Read API

Algorithm ?? presents the functions of the reader API.

```

1 // Reads a single line from a segment
2 int shmrbuf_read(struct shmrbuf_reader_arg_t * args,
3                   void * rbuf,
4                   uint16_t bufsize,
5                   uint8_t segment_id);
6
7 // Reads multiple lines from a segment
8 int shmrbuf_readv(struct shmrbuf_reader_arg_t * args,
9                   struct iovec * iovecs,
10                  uint16_t vsize,
11                  uint16_t bufsize,
12                  uint8_t segment_id);
13
14 // Reads a line from a segment out of a specified range.
15 int shmrbuf_read_rng(struct shmrbuf_reader_arg_t * args,
16                      void * rbuf,
17                      uint16_t bufsize,
18                      uint8_t lower,
19                      uint8_t upper,
20                      bool * wsteal);
21
22 // Reads multiple lines from a range of segments
23 int shmrbuf_readv_rng(struct shmrbuf_reader_arg_t * args,
24                      struct iovec * iovecs,
25                      uint16_t vsize,
26                      uint16_t bufsize,
27                      uint8_t lower,
28                      uint8_t upper,
29                      uint16_t * wsteal);

```

Algorithm 3.8: Read API for the shared memory ring buffer.

shmrbuf_read is the corresponding read operation to shmrbuf_write. It reads a single from the specified segment into the provided buffer. If the provided buffer size is exceeded by the size of the line, the read call fails, hence, the buffer should always at least have maximum line size, specified in the global header. Upon a call to shmrbuf_read, the function atomically loads the

write index for the segment. If the position of the write index differs from that of the readers read index, the line pointed to by the reader index will be copied to the external buffer and the read index will be atomically updated to its new position. On success, `shmrbuf_read` returns the (maximum) line size or zero, if the buffer is empty⁴. Analogous to the write API, a vectored read function exists in `shmrbuf_readv`, which should incur the same performance benefits. Instead of reading a single line, `vsizes` lines are copied to the buffers specified by `in` corresponding `iovec` structures in `iovecs`. Unlike the write API, all read operations called on the same segment by threads within the same processes are thread safe. At the beginning of each read operation, the mutex lock in the corresponding `shmrbuf_seg_rhdr_t` structure is claimed and release after the read index has been updated. For performance reasons, frequent locking of a segment by different reading threads should be avoided and threads should ideally be assigned segments that they access exclusively. The read API contains two further functions. `shmrbuf_read_rng` is a convenience wrapper around `shmrbuf_read`. It is useful for the case, that the reading application uses fewer reading threads than there are segments in the buffer. `shmrbuf_read_rng` allow the specification a segment range through the `upper` and `lower` parameters. When calling `shmrbuf_read_rng`, the function will iterate over all segments in the range in a round-robin fashion and perform a read call, until the first non empty segment is found or one cycle is completed. The iteration index is persistent across function calls, ensuring that subsequent calls will continue the round-robin at the same position. This allows starvation-free reading from the specified segment range. The boolean pointer `wsteal` allows for the option of workload stealing. If `wsteal` is non null and all segments within the specified range are empty, the function will additionally iterate over all segments outside of the specified range until a line has been read successfully or all segments have been checked. If a line was read from outside the specified range, `wsteal` will be set to true. The intention of the workload stealing feature is to mitigate scenarios, where writing operation are not equally distributed across segments. If a reader, for instance, assigns two reading threads to one segments each, but only one of the segments is being written to, then one reading thread would be idle, while the other has to handle all reading operations. Workload stealing mitigates this, by having threads read across their assigned boundaries, if they would otherwise be idle. `shmrbuf_readv_rng` is the last function in the read API and corresponds to `shmrbuf_read_rng` with the only difference, that multiple lines can be read from each segment, up to the number specified by `vsizes`. The `wsteal` parameter is also an integer instead of a boolean pointer and will be set to the number of lines read from outside the specified range, if `wsteal` is non null.

3.5 Proof-of-Concept IPS

The following section presents the design and implementation of the proof of concept IPS, for testing the IPC architecture covered in the past sections. In order to be comparable, the IPS will be closely modelled after Fail2ban. The goal is however not to fully reimplement Fail2bans full set of features, as that would go beyond the scope of this thesis. Instead, a minimal set of features

⁴In its current implementation, the buffer allows any type of binary data to be copied to a line. This is a problem for determining the actual length of a string in the line buffer, since the terminating zero byte could still be part of the payload. Therefore, the maximum line length is always returned. Restricting the buffer to non zero characters, or moving to a variable line length design, would enable the function to determine the length of the string contained in the line buffer, making the return value more meaningful.

3 Design & Implementation

will be supported, the allows for the replication of the experiment covered in section 2.1.1. The requirements for the **Poc!** (**Poc!**) are as follows:

- Ability to monitor an application log, via traditional logfile parsing and the new IPC solution.
- Ability to parse log messages with a customly definable regular expression.
- Ability to create filter rules for a clients IP addresses for both Internet Protocol Version 4 (IPv4) & IPv6.
- Support for a ban limit, i.e. a customly definable number of matches per client before a ban is executed.
- Support for a ban time, i.e. a customly definable duration for the ban of a clients IP address.

Figure 3.3 illustrates the proposed design for the PoC, which will from hereon be referred to as Simplefail2ban. The source file for the implementation of Simplefail2ban is `simplefailban.c` in the `src/programs` directory of the thesis Git repository [30].

The functionality of the application will be separated into two classes of threads. A variable number of “banning” threads and a single “unbanning” thread. The banning threads are tasked with monitoring the application log. The routinely call a read function, to check if new messages have been added to the log. The read function varies, depending on the type of IPC architecture being used. For multi-threaded use, the read function has to be thread-safe and should ideally be lock free, to allow for optimal performance. When using the shared memory ring buffer as the source for log messages, `shmrbuf_readyv_rng` is used, where the buffer segments are equally distributed among the banning threads. For traditional logfile parsing, a custom read function was implemented, on the basis of the asynchronous IO library liburing introduced in 2.3.2.

```
1 // Reads a single line from a file (buffered)
2 int uring_getline(struct file_io_t * fio_arg,
3                   char ** lineptr);
4
5 // Reads multiple lines from a file (buffered)
6 int uring_getlines(struct file_io_t * fio_arg,
7                     struct iovec * ivoecs,
8                     uint16_t vszie,
9                     uint16_t bufsize);
```

Algorithm 3.9: Function signatures for the io_uring based getline functions.

?? displays the function signatures. `uring_getline` was modelled after the `getline` function from the `stdio.h` library within the C standard library [32]. The function is passed a pointer to a `file_io_t` structure which contains file descriptor for the designated file, as well two buffers for the data being read. When the function is called, both buffers are checked for lines from an earlier function call. If both buffers are empty, a liburing based read operation is initiated. The read data is then tokenized into lines via a linear search for newline characters. If the read

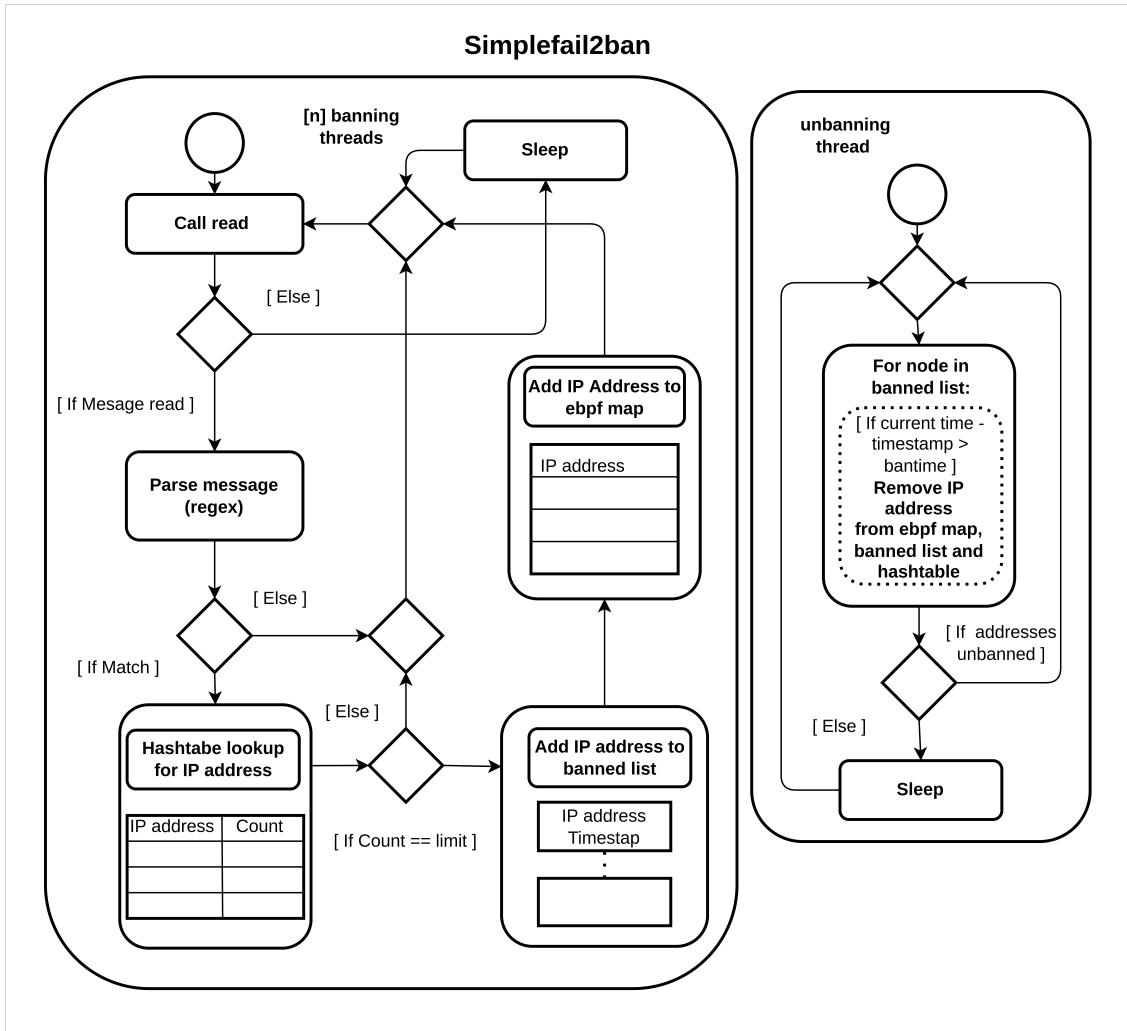


Figure 3.3: Activity diagram for the proof-of-concept IPS implementation. A variable number of “banning threads” receive log messages from a host and parse them with a predefined regular expressions. For messages that match the expression, the clients IP address is extracted from the log message and added to a hashtable, that keeps count of the number of matches per address. If the count reaches the configured limit, the address is added to the list of banned addresses with a current timestamp and inserted into the eBPF map. One “unbanning thread” routinely iterates through the banned list and checks, if a clients bantime hast elapsed. Clients with an elapsed bantime are removed from the eBPF map, banned list and hashtable.

3 Design & Implementation

operation fills the entire buffer, another read operation is scheduled for the second buffer. The return of the read operation is not awaited and will only be checked on a subsequent function call. The idea is, that, in anticipation of subsequent function calls, data is being asynchronously read ahead, so that for most function calls, the function can resort to the buffer instead of reading from the actual file. This should hopefully be more performant than regular blocking calls to read. The file offset for the read operation is tracked in the `file_io_t` structure and updated after each read. `uring_getlines` works analogous to `uring_getline`, with the difference that up to `vsize` lines can be read in a single function call. The lines and their size in byte are copied to the corresponding `iovec` structure in `iovecs`. `uring_getline` and `uring_getlines` are not build for thread-safety, hence, the logfile based version of Simplefail2ban only supports single threaded monitoring⁵.

If a banning thread has successfully read one or multiple log messages, the messages are then matched against a regular expressions, which can be defined at the start of the application. For Regular Expression (Regex)-matching the regular expressions engine Hyperscan, introduced in section 2.3.1, is used. Hyperscan is used in multi-pattern matching mode, scanning for both the provided Regex as well as expressions for IPv4 and IPv6, to extract the clients IP address from the log message. If the message was successfully matched against the provided Regex, the clients IP address identified by the IP-Regex will be translated to its binary form for further processing. If the scan does not find a match, the processing moves on to the next log message. Alternatively, Simplefail2ban can also be configured to directly translate the entire log message to an IP address, without regular expression matching. This is not explicitly referenced in 3.3 and was only implemented as reference, to measure the impact of Regex-matching on the overall performance of the application.

Once a clients binary IP address has been determined (either through successful matching or direct translation), a hash table lookup for the address is performed. The purpose if this is to determine, if a client has reached the ban limit, which is configured at the start of the application. The hash table stores key value pairs, where the key can be a IPv4 or IPv6 address in binary form and the value is a counter, indicating the number of times the key has been queried. A custom hash table was implemented as a static library, the sources files for which are `ip_hashtable.h` and `ip_hashtable.c` in the `src/lib` directory of the thesis Git repository [30].

?? displays the structure used for storing a single entry in the hash table. The base table consist of an array of bins, with a default size of 6000011⁶. When an ip address is inserted into the table, an index within the array is determined by calculated the value of a hash function modulo the size of the array. The address and its associated counter value are then stored in the bin at the determined index, if it is empty. The hash function used for the implementation is spookyc, which is a C implementation of the spooky hash function by Bob Jenkins [33]. If the bin at the determined is not empty, a collision occurs. Collisions are handled via a linked list, where the bin in the base array constitutes the first node of the list.

Table ?? presents experimental results for the collisions performance of the hash function. There appear to be no significant differences in collision behaviors between IPv4 and IPv6 addresses. For 6000011 insertions (size of the base table), a little more than a third of all insertions

⁵It its unclear, wether multi-threaded file reading would provide a performance benefit, since the read operations would have to be serialized by the operating system. Hence the decision to not support multi-threading.

⁶The array size was purposefully chosen as a prime number, to allow for a more equal spread among the bins.

```

1 // Struct to store a single hashtable entry
2 struct ip_hashbin_t
3 {
4     void * key;
5     int domain;
6     uint32_t count;
7     pthread_mutex_t lock;
8     struct ip_hashbin_t * next;
9
10 };

```

Algorithm 3.10: Structure for storing a single entry in the IP hash table. The `key` pointer points to the binary address, the size of which depends on whether the `domain` value is `AF_INET` or `AF_INET6`.

Number of Insertions	Collisions IPv4 [%]	Collisions IPv6 [%]	Collisions IPv4 & IPv6 [%]
65534	5.33	5.29	5.28
131068	10.17	10.17	10.15
600011	36.81	36.77	36.8

Table 3.1: Percentage of hash collisions by key type for different numbers of insertions. The results were obtained with `hashfunc_benchmark.c` in the `src/utilities` directory of the thesis Git repository [30].

resulted in a collision. The hash table is global to the application, since different banning threads may handle log messages for the same client and need to synchronize their match count. To ensure thread-safe operations on the hash table, locking via mutexes is used. The granularity of the locking is at bin level, hence, parallel operations on the table are possible if different bins are accessed.

If the hash table lookup returns a count that is equal to the ban limit, the banning thread facilitates a ban action, analogous to Fail2ban. To implement IP address filtering, the eBPF program developed by Florian Mikolajczak [1] as described in 2.1.1, was used. The program can be loaded onto a network interface and is executed event based on incoming packets⁷. To determine which packets should be dropped, the program uses maps containing IP addresses, which are pinned to the eBPF file system. At the start of Simplefail2ban, the eBPF program will be loaded onto a configured interface. The function used for loading and unloading the eBPF program and corresponding maps are slightly adapted from the implementation by Florian Mikolajczak and can be found in `ebpf_helpers.h` and `ebpf_helpers.c` in the `src/lib` directory of the thesis Git repository [30]. The ban action executed by the banning thread consists of adding a client's binary IP address to the corresponding eBPF map. There are two distinct maps being used for IPv4 and IPv6 addresses. Once the address has been added to the map, the eBPF program will drop all incoming packets from that address, for as long as the address is contained in the map. Addi-

⁷For a more thorough and detailed explanation, see [1]

tionally, the banning thread adds the IP address to a linked list (referred to as banned list in 3.3) together with a current timestamp. The linked list is used to store client that are currently banned. The source files for the linked list implementation are `ip_llist.h` and `ip_hllist.c` and can be found in the `src/lib` directory of the thesis Git repository [30]. The unbanning thread will routinely iterate over the list and checked the amount of time that has passed since the timestamp. If the configured ban time has elapsed, the unbanning thread will unban the client. To unban a client, the unbanning thread removes its entry in the linked list as well as the entry in the eBPF map. Finally, the address entry in the hash table is removed. This resets the matching count, so a client can be banned, if the ban limit is reached again.

3.6 Test Application

To evaluate the IPC architecture in conjunction with Simplefail2ban, a test application is needed, which utilizes the IPC API to transmit log messages. For this purpose, a test server was developed, the source file for which is `udp_server.c` in the `src/programs` directory of the thesis Git repository [30]. Alternatively, the IPC architecture could have been integrated into a real application, like the BIND server used in the previous Fail2ban measurements by Florian Mikolajczak. While this would have provided a more realistic basis for the evaluation, the associated implementation effort was deemed beyond the scope of this thesis. The test server serves as a stand in for a real UDP based application such as BIND. It listens on a configurable port, and replies to incoming packets with a single one byte payload UDP packet. Based on the first payload byte of the received request, the server decides whether to write a log string for the requesting client⁸. The log string contains the current date, time and IP address of the client, as well as a descriptive message and is written to either a logfile or the shared memory ring buffer. Alternatively, the test server can be configured to only log a clients IP address. The test server uses multiple threads to listen for incoming packets and is designed to handle a large amount of requests and logging operation per second, in order to not be a bottleneck for the evaluation of Simplefail2ban. Perf [31] evaluation of the test server revealed, that about 20% of the execution time on the application side was spent translating IP addresses to string form. To facilitates fast creation of log strings, a custom function was written for the transformation of a binary IP address to string form.

Function	Execution Time IPv4 [Seconds]	Execution Time IPv6 [Seconds]
<code>inet_ntop</code>	1.29	3.98
<code>ip_to_str</code>	0.21	0.53

Table 3.2: Performance evaluation for binary IP address to string conversion. The evaluated functions are `inet_ntop` from `arpa/inet.h` in the C standard library and a custom function `ip_to_str`. The corresponding source files are `ip_to_str.h` and `ip_to_str.c` in the `src/lib` directory and the evaluation was conducted with `ip_string_benchmark.c` in the `src/utilities` directory of the thesis Git repository [30]

Table 3.2 summarizes the evaluation results for the custom function `ip_to_str` and the standard library function `inet_ntop`. `ip_to_str` is on average about 6 times faster for translating IPv4

⁸The byte to trigger logging can be determined with the `INVALID_PAYLOAD` macro in `udp_server.c`.

addresses and about 7 times for IPv6. Since the test server can write up to a million log messages per second, overall performance should be improved by the use of `ip_to_str`⁹.

3.7 Other Applications

Two other applications were developed as part of this thesis. To test the multi reader capability of the proposed shared memory architecture, an application serving as an additional reader was implemented. The application models a log aggregator such as Logstash (discussed in 2.2.1) and writes the log messages contained in the buffer to a configurable logfile. The source file for this application is `simplelogstash.c` in the `src/utilities` directory of the thesis Git repository [30]. The second application is a utility for inspection and debugging the shared memory ring buffer. It allows the inspection of the header as well as the display of load statistics on the individual segments and the entire buffer. The source file is `poll_rbuf.c` in the `src/utilities` directory of the thesis Git repository [30].

⁹This could be further evaluated by comparing overall difference of the server for both functions, but I unfortunately did not have the time to implement further test.

4 Evaluation

The following section presents the performance evaluation of the proof of concept IPS Simplefail2ban, in conjunction with the test server. First, an overview over the test environment and the experimental design is given and the Fail2Ban performance issues, discovered by Florian Mikolajczak, are replicated for the test server. Subsequently, the result for Simplefail2ban in both the logfile and shared memory based variants are presented. The section concludes with an evaluation of different features of the shared memory implementation.

4.1 Test Environment

To conduct the evaluation, two machines with an identical hardware and software configuration were used. The specific hardware and software version are listed in Table 4.1. One machine served as the dut, running the test application, as well as Fail2Ban / Simplefail2ban. The second machine was used for traffic generation with TRex.

Table 4.1: Hardware and Software parameters for the test environment. The measurements were conducted on two identical machines, where one machine served as the Device under Test (DUT), running the test application and Fail2Ban / Simplefail2ban, while the other generated test traffic with trex.

Hardware	
CPU	16 × Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz
NIC	Mellanox ConnectX-6 100GbE
RAM	128GB
Software	
OS	Debian 11
Kernel	6.1.0-0
NIC Driver	mlx5_core, 6.1.0-0
Fail2Ban	0.11.2
TRex	3.02

4.2 Experimental Design

The experimental design is closely oriented at experiment 1 conducted by Florian Mikolajczak in [1], in order to provide comparable results. In the experiment, a **DOS!** attack on a server under a normal workload is simulated, through a stream of valid a request and a significantly larger stream of invalid request. The original experiment used a BIND DNS server as the target of the **DOS!** attack, which triggered log messages for clients exceeding a certain rate limit. For the following evaluation, the test server introduced in 3.6 will serve as the target, differentiating valid and invalid request by the sent payload¹. While this way of identifying unwanted request is not necessarily realistic, it is efficient and allows the test application to quickly produce log messages.

The goal of this evaluation is to test, how Simplefail2ban performs for the test scenario described above. Following the guidelines proposed by Jain [34] performance metrics and factors were defined and system / workload parameters identified:

Performance Metrics

- Total amount of invalid requests dropped (number of packets)
- Total amount of invalid requests dropped, relative to the total amount of invalid request sent (percentage)
- Amount of log messages processed by Simplefail2ban, relative the amount of log messages sent by the test server (percentage)
- Central Processing Unit (CPU) utilization of Simplefail2ban (percentage)

Parameters

- Hardware and software parameters of the environment listed above (4.1). Especially:
 - CPU: 16 cores, no hyper-threading enabled
 - Network Interface Card (NIC): MTU 1500 bytes
 - TRex: One interface, 30 threads
- Number of entries in eBPF maps for IPv4 & IPv6 : 1000000
- Number of receiving threads used by the test server : 16
- Duration of the measurement : 300 seconds
- Amount of valid traffic sent : 10000 PPS
- Amount of clients sending valid traffic : 254
- Simplefail2ban parameters
 - Number of banning thread used : 4 (shared memory), 1 (logfile)
 - Number of hash table bins used : 6000011
 - Ban threshold for clients : 3

¹Request are considered invalid, if the first byte if the payload has the unsigned value 42 (ASCII Letter B)

- Ban time for clients : 30 Seconds
- Line count for shared memory buffer segments : 100000
- Segment count for shared memory buffer : 16
- Buffer size for io_uring getline : 1024

Factors & Levels

- Amount of invalid traffic sent : 100k, 1m, 10m PPS
- Amount of clients sending invalid traffic : 65534, 131086
- IP stack: IPv4, IPv6, IPv4 / IPv6 mixed
- Simplefail2ban factors
 - IPC Type: Logfile / Shared Memory
 - Overwrite / No Overwrite (shared memory only)
 - 2nd Reader / No 2nd Reader (shared memory only)
 - Workload stealing / No workload stealing (shared memory only)
 - Regex matching / No Regex matching

The measurements are conducted with TRex scripts, which define a packet stream, that can be customized to generate traffic in accordance with the factor levels listed above. The scripts are adapted from the original TRex scripts used by Florian Mikolajczak [1] and can be found in the `src/scripts/traffic_gen` directory of the thesis Git repository [30]. For measuring traffic rates as well as the number of dropped packets, an adapted version of the `xdp_ddos01_blacklist_cmdline` program from the implementation of Florian Mikolajczak was used [1]. The program repeatedly polls maps that are continuously updated by the loaded eBPF program with the number of passed and dropped packets.

Experiment 1: Fail2ban Replication

The first experiment experiment is a replication measurement, meant to replicate the results obtained by Florian Mikolajczak [1] for the new test environment. This is done, in order to verify, that the performance issues persist the new test application as well as the upgraded hardware, compared to when the experiment was first conducted. Two constant streams of UDP packets will be sent to the test server, one of which contains invalid payloads that trigger log messages and should eventually result in the banning of the clients. The chosen factor levels are the same as in the original experiment, at 50000 valid requests per second and 1000000 invalid request from 254 different clients. The only difference is the reduction of the ban time from 180 to 60 seconds and the overall measurement time, from 900 to 300 seconds.

As a variation, a seconds Fail2ban measurement is conducted, which lowers the traffic invalid and valid traffic rates to 10000 PPS but increases the number of clients sending invalid request to 1016. This experiment aims at evaluating the scaling of Fail2bans performance in terms of the number of clients.

Experiment 2: Simplefail2ban Logfile

Experiment 2 evaluates the performance the logfile-based variant of Simplefail2ban. The setup follows that of Experiment 1, but the parameters are instead chosen as listed in the Parameters section above. The factors that are varied are the amount of invalid traffic, the amount of clients sending invalid requests and the IP-stack used for the UDP traffic. Simplefail2ban is used with Regex matching enabled. In total, 9 measurement are conducted.

Experiment 3: Simplefail2ban Shared Memory

Experiment 3 mirrors experiment 2, except that the shared-memory based variant of Simplefail2ban of Simplefail2ban is evaluated. For the measurement, workload stealing and overwrite are disabled and no second reader is attached to the shared memory ring buffer. Simplefail2ban is used with Regex matching enabled. In total, 9 measurement are conducted.

Experiment 4: Shared Memory Features

The final experiment evaluates the different features of the shared memory implementation. All measurements are conducted with the highest load scenario, since performance differences are expected to be most pronounced there. Hence, the invalid traffic rate is chosen at 10m PPS, the amount of clients sending invalid traffic at 131068 with a mixed IPv4 & IPv6 stack. The factors that are varied (individually) are overwrite, adding a second reader, workload stealing and disabling Regex matching. In total, 4 measurements are conducted.

In the following sections, the results of experiments 1 through 4 are presented.

4.3 Experiment 1: Fail2ban Replication

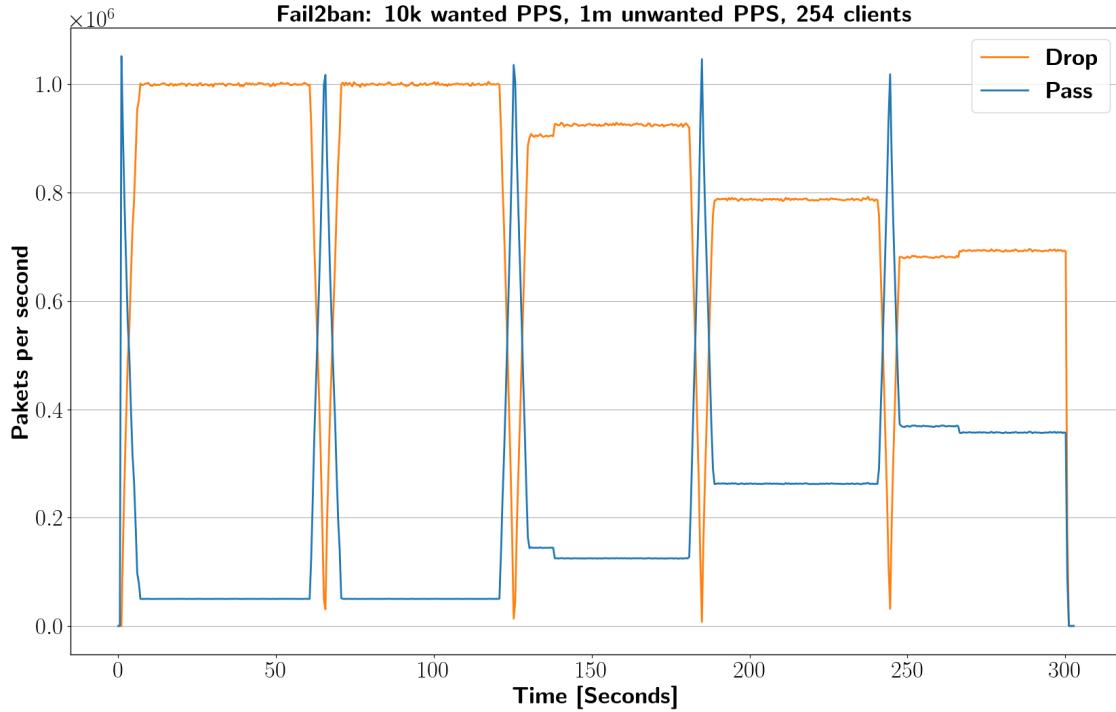


Figure 4.1: Fail2ban Replication: 1 million invalid PPS from 254 Clients, 50 thousand valid PPS

Figure 4.1 displays the results of the replication measurement. While the results differ from those presented in ??, a similar pattern can be observed overall. Initially, Fail2ban performs as expected, banning all unwanted traffic for the specified duration. This even holds for two full ban cycles, which is the first notable difference from previous measurements. After the second ban cycle, the drop rate falls below the expected level and does not recover after subsequent ban cycles. The reason for this behavior is unclear. One hypothesis in the original work by Florian Mikolajczak [1] was, that the initial ban cycle is successful due to TRex slow build up of the traffic rate. However, this does not explain the success of the second ban cycle, where the traffic rate is at full capacity. Another possible explanation could be, that Fail2bans responsiveness fluctuates due to internal or external factors. If the responsiveness is low at the end of a ban cycle, Fail2ban does not ban enough clients in time and is overwhelmed by the influx of log messages. However, this remains speculation and warrants further investigation into Fail2bans performance for future work.

Figure 4.2 presents the results for the second Fail2ban measurement. Here, the result are much more in line with expectation. While Fail2ban manages to ban and unban all clients for repeated ban cycles, it takes more than 20 seconds for both banning and unbanning. The measurement was repeated for a smaller number of clients, which resulted in significantly better performance. This

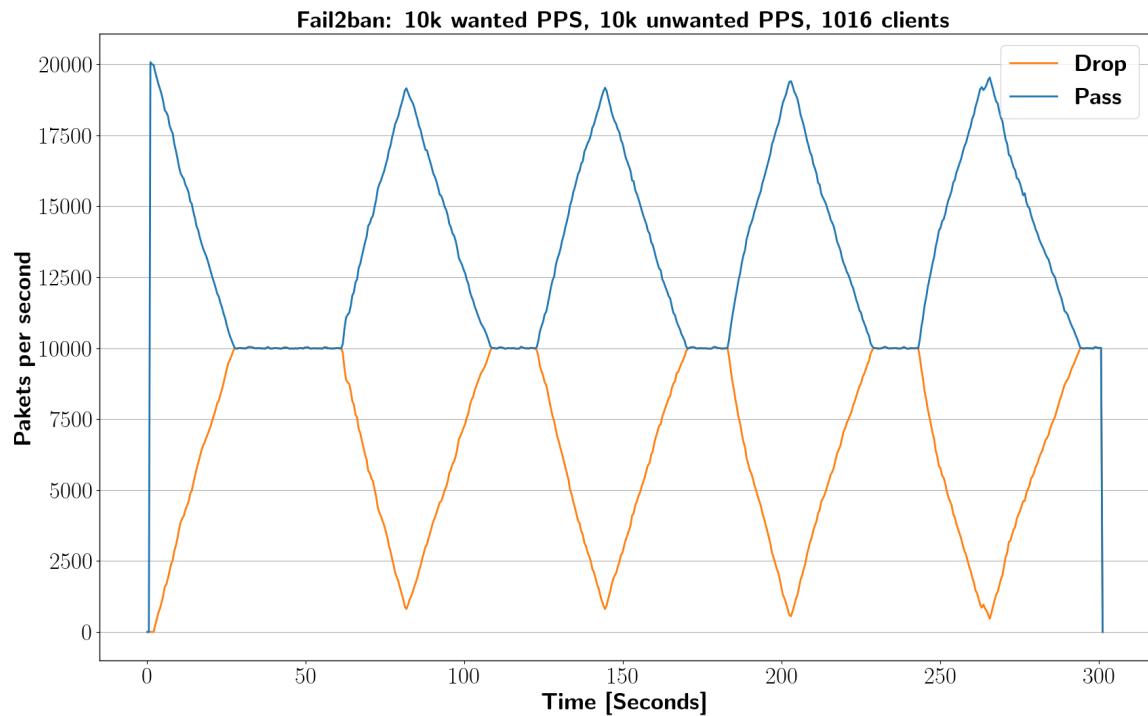


Figure 4.2: Fail2ban Replication: 10 thousand invalid PPS from 1016 Clients, 10 thousand valid PPS

illustrates, that the number of different clients managed by Fail2ban has a strong impact on the performance of Fail2ban, even at low traffic rates.

Overall, the replication managed to show, that Fail2bans performance issues still persist in the new test environment. Both the rate of traffic as well as the number of clients appear to be significant factors, for which Fail2bans performance scales poorly.

4.4 Simplefail2ban Measurements

The following sections present the results for the Simplefail2ban evaluation. To make the reading experience more concise, figures presenting the results for the dedicated IPv4 and IPv6 measurement have been moved to the annex. However, their results will still be discussed as part of this chapter.

Disclaimer

Upon renewed replication of the measurement, some discrepancies were discovered, which had not been observed when the initial measurements and replication were conducted. More specifically, the performance measured for Simplefail2ban appears to vary, which is most pronounced for the 10 million PPS measurements. On some replication attempts the unbanning and renewed banning of clients took about 5 to 10 seconds longer than in previous measurement, resulting in a lower rate of dropped packets relative to the expected drop rate. I have not been able to identify, whether this is caused by a change in the test environment or is simply a characteristic of the implementation. Unfortunately, I lack the time to conduct a proper variance assessment in time for submission of this thesis. Hence, the disclaimer is issued, that the results presented in the following may not be fully representative of the average system performance. A more thorough investigation of this problem will be provided at the thesis presentation.

Result overview

For the IPv4 and IPv6 measurements 65534 clients were used for sending invalid traffic, while the mixed IPv4 & IPv6 measurement was conducted with 131068 clients. The total packets, packets dropped and log messages column in the following figures are absolute values, aggregated across the entire measurement time. Total packets refers to the sum of both dropped and passed packets. Log messages refers to the number of messages processed by Simplefail2ban². Relative drop refers to the number of dropped packets, relative to the ideally expected amount of dropped packets. This number is given by: total dropped packets / (total packets – experiment duration * valid traffic rate – number of ban cycles * ban limit). Finally, CPU refers to the CPU usage for Simplefail2ban during the entire measurement, as given by top.

4.4.1 Experiment 2: Logfile Variant

Figure 4.3 and figures ??, ?? in the annex, present the results for the logfile based Simplefail2ban at a traffic rate of 100 thousand invalid PPS. Overall, the results are very similar for all three measurements and generally, over 99% percentage of invalid traffic is dropped. The visible gap between ban cycles can be explained by the test setup. TRex iterates over all clients address sequentially when sending traffic and the ban limit requires each client to at least send at least three packets before being banned. At a rate of 100k invalid request a second, this can take several seconds, dependant on the number of clients. One slightly unexpected observation is, that the unbanning of clients does not always appear to occur at the same time, which increases for subsequent ban cycles. A possible explanation for this is, that the timeout value for the unbanning thread was chosen to coarsely at 0.5 seconds. When the unbanning thread iterates over the

²For nearly all measurement, this number was the exact same as the number of log messages written by the test server. If not, it will be explicitly stated otherwise.

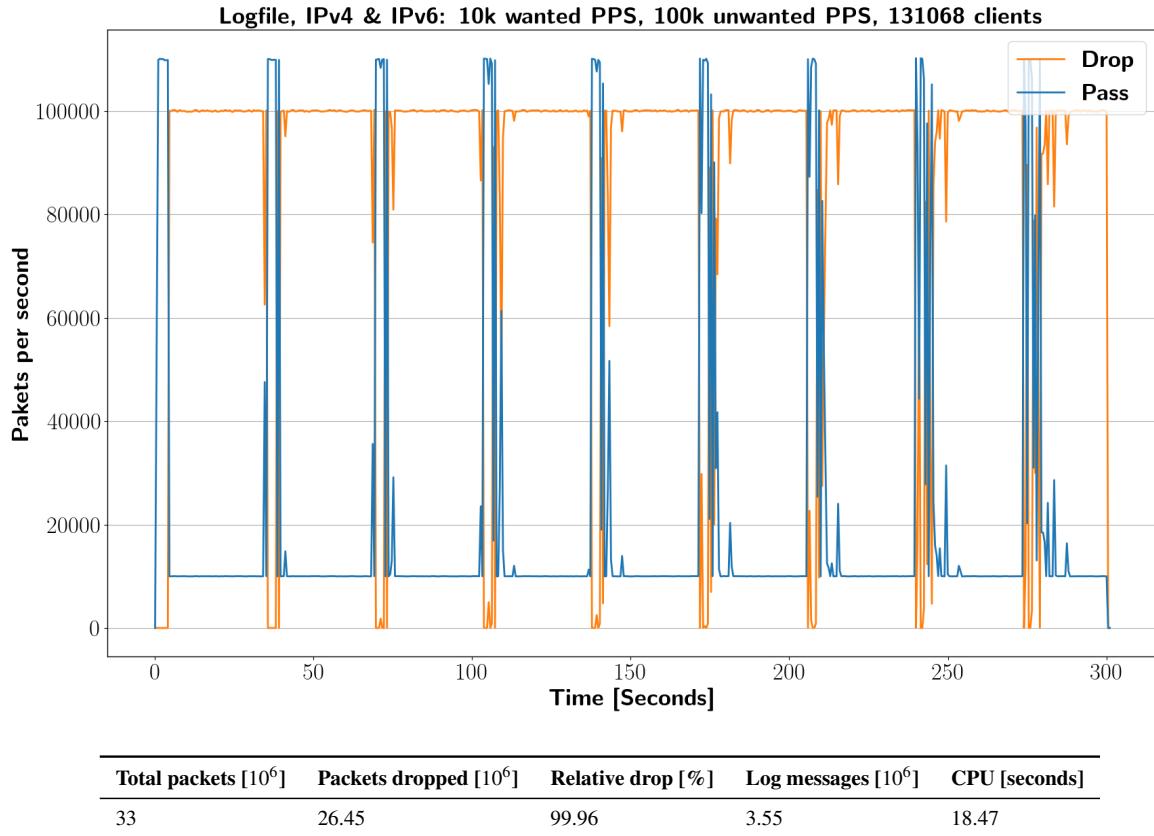


Figure 4.3: Some text

list of banned clients without performing a unbanning action, the thread will sleep for the specified timeout before checking the list again. If poorly timed, this could lead to gaps between the unbanning of clients, which accumulate over time. Since the same behavior is observed in later measurement, this issue should be investigated, if the development of Simplefail2ban is continued.

Figure 4.4 and figures ??, ?? in the annex, present the results for the logfile based Simplefail2ban at a traffic rate of 1 million invalid PPS. Again, the result are very similar for all three measurement. Compared to the results at 100 thousand PPS, the relative drops by about one percentage point across the measurements to around 98%. The relative drop for the joint IPv4 & IPv6 measurement is also slightly worse by about half a percentage point, than the other two measurements, which can be explained by the fact, that the double the amount clients need to be banned and unbanned. The pure IPv4 measurement also slightly outperforms pure IPv6 in terms of relative drop rate. This is likely the result of longer log messages due to the length of the IPv6, which is more taxing for both the file reading as well as the Regex matching.

Finally, figure 4.5 and figures ??, ?? in the annex, present the results for the logfile based Simplefail2ban at a traffic rate of 10 million invalid PPS. Compared to the previous result, overall

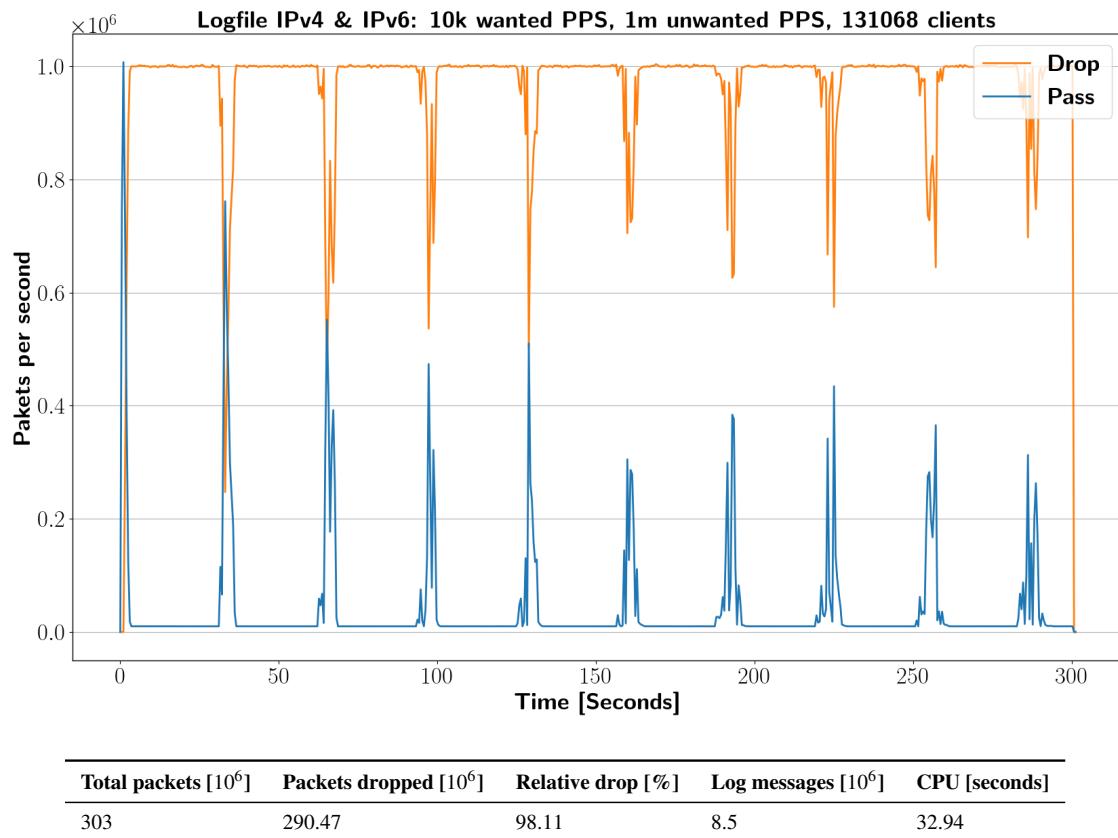


Figure 4.4: Some text

performance drops drastically. The best result is achieved for the purpose IPv4 traffic, which maintains a high relative drop rate of 96.72 %, even though CPU usage and the amount of messages processes are about 6 times higher than in 1 million PPS measurement. For IPv6 and the joint traffic, result are much worse, as only around 69% and 67% of the expected traffic is dropped. The results are similar to those of Fail2ban in figure ???. Simplefail2ban appears to be overwhelmed with the processing of log messages at a certain point and does not manage to recover before the end of the ban cycle. For the IPv6 measurement Simplefail2ban managed to process only about 89% of the total log messages written by the server. For the joint measurement the number is significantly lower at around 64%. Overall the result show, that, even though the threshold appears to be larger, the same issues of file based log message parsing persist for Simplefail2ban.

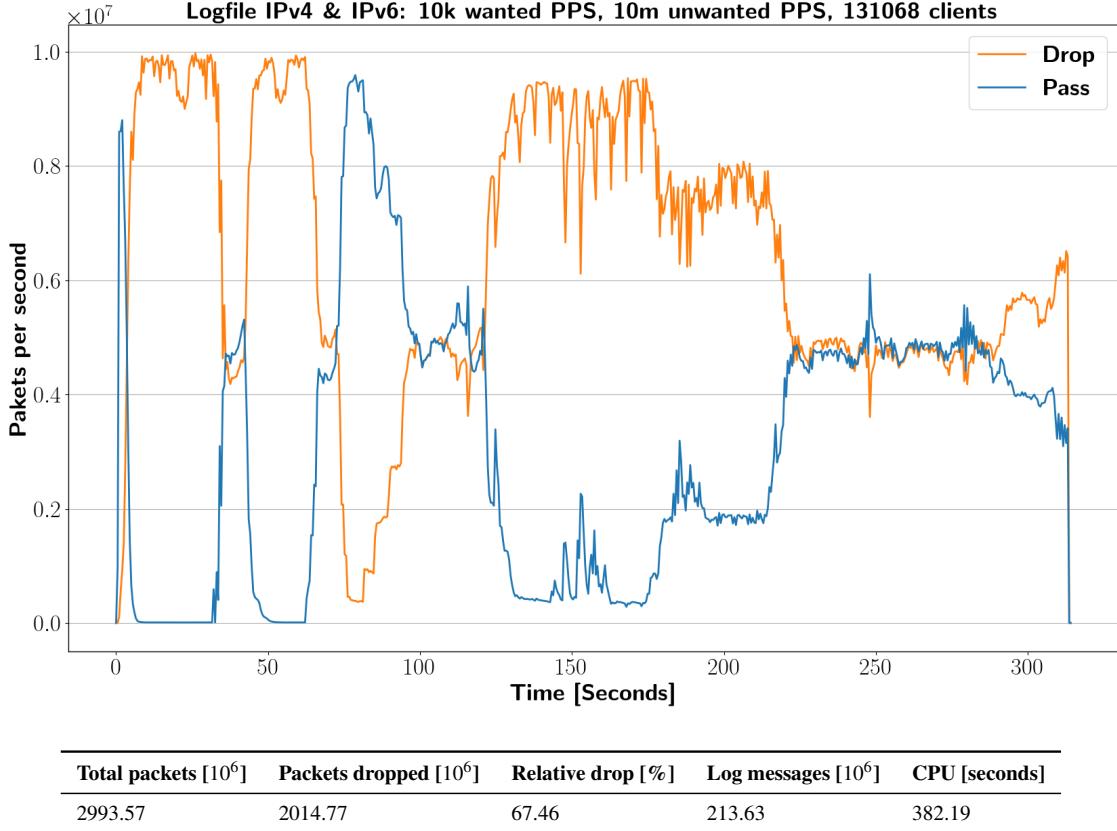


Figure 4.5: Some text

4.4.2 Experiment 3: Shared Memory Variant

Figure ?? and figures ??, ?? in the annex, present the results for the logfile based Simplefail2ban at a traffic rate of 100 thousand invalid PPS. All three measurements show similar results, which are also similar to the result measured for the logfile-based version of Simplefail2ban. Again a relative drop rate of over 99% percent is achieved across the board. The only noticeable difference to logfile-based result is, that the CPU usage appear to be slightly higher. However, this observation is skewed by the fact, that the execution time of the kernel performing the io_uring based reads is not included in the statistic.

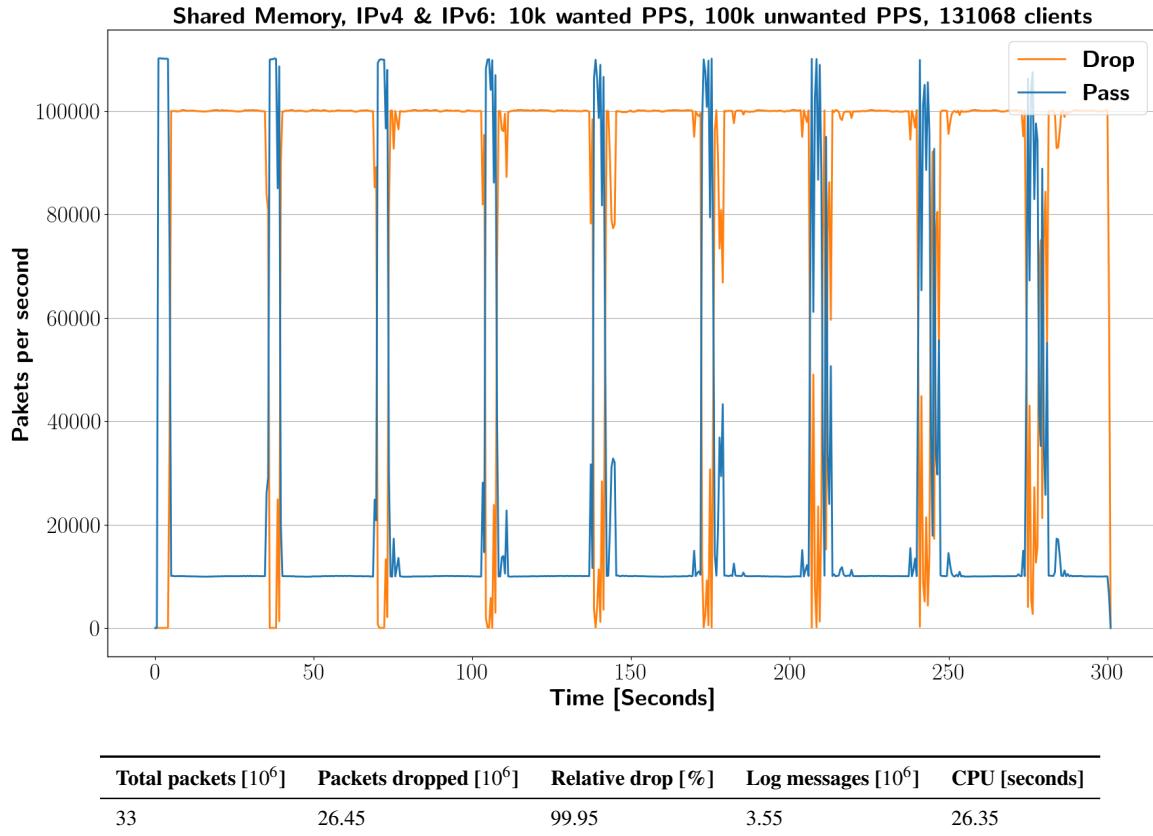


Figure 4.6: Some text

Figure ?? and figures ??, ?? in the annex, present the results for the logfile based Simple-fail2ban at a traffic rate of 1 million invalid PPS. Again, the results are very similar, both between the measurements, as well as compared to the logfile-based measurements. Overall, the shared memory based Fail2ban performs slightly better across the board, with a relative drop rate of around 99%. Interestingly, there appears to be no significant performance difference between the pure IPv4 and IPv6 and the joint measurements, even though the latter has to process double the amount of clients.

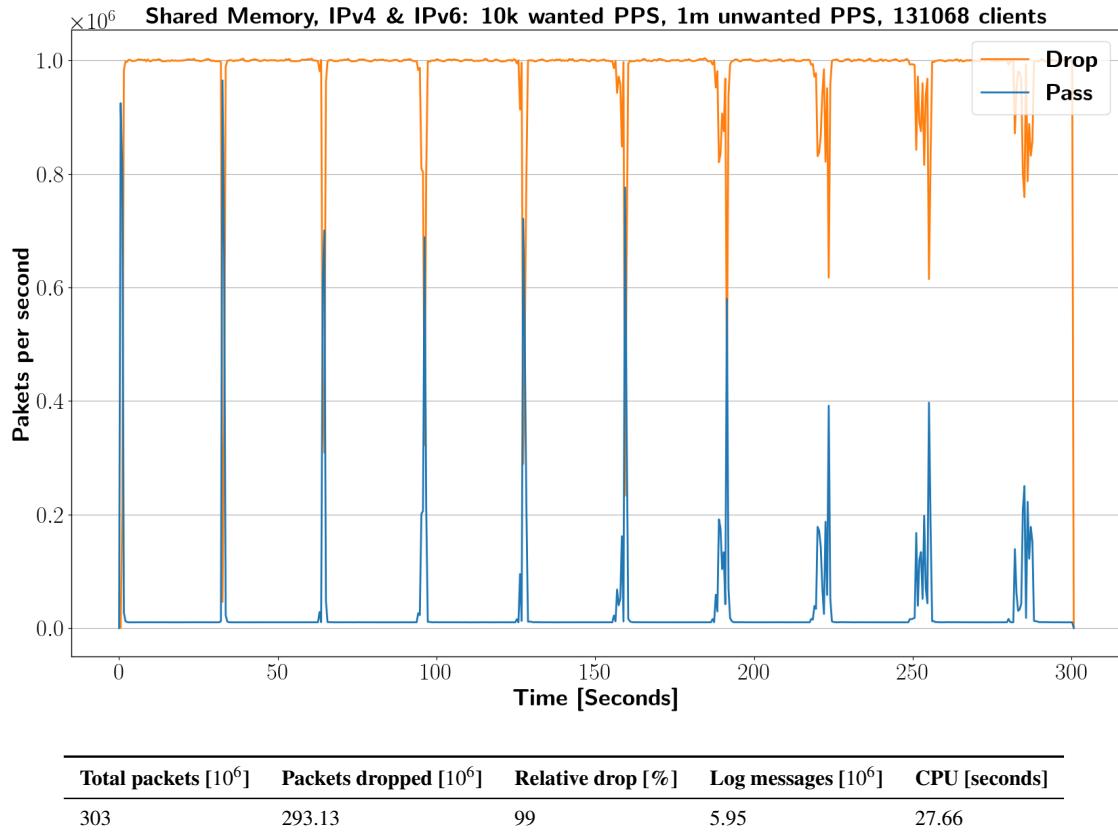


Figure 4.7: Some text

Figure ?? and figures ??, ?? in the annex, present the results for the logfile based Simplefail2ban at a traffic rate of 10 million invalid PPS. Compared to the previous logfile-based measurements, performance is a lot better. Both the IPv6 and joint measurements achieved a relative drop rate of more than 98%. Counterintuitively, the pure IPv4 measurement performed the worst, at only 96.72 %. The reason for this is not entirely clear, but it may be further indication that the performance of Simplefail2ban varies more than the initial measurements indicated. Hence, further replication is necessary to provide a more stable estimation of the expected performance.

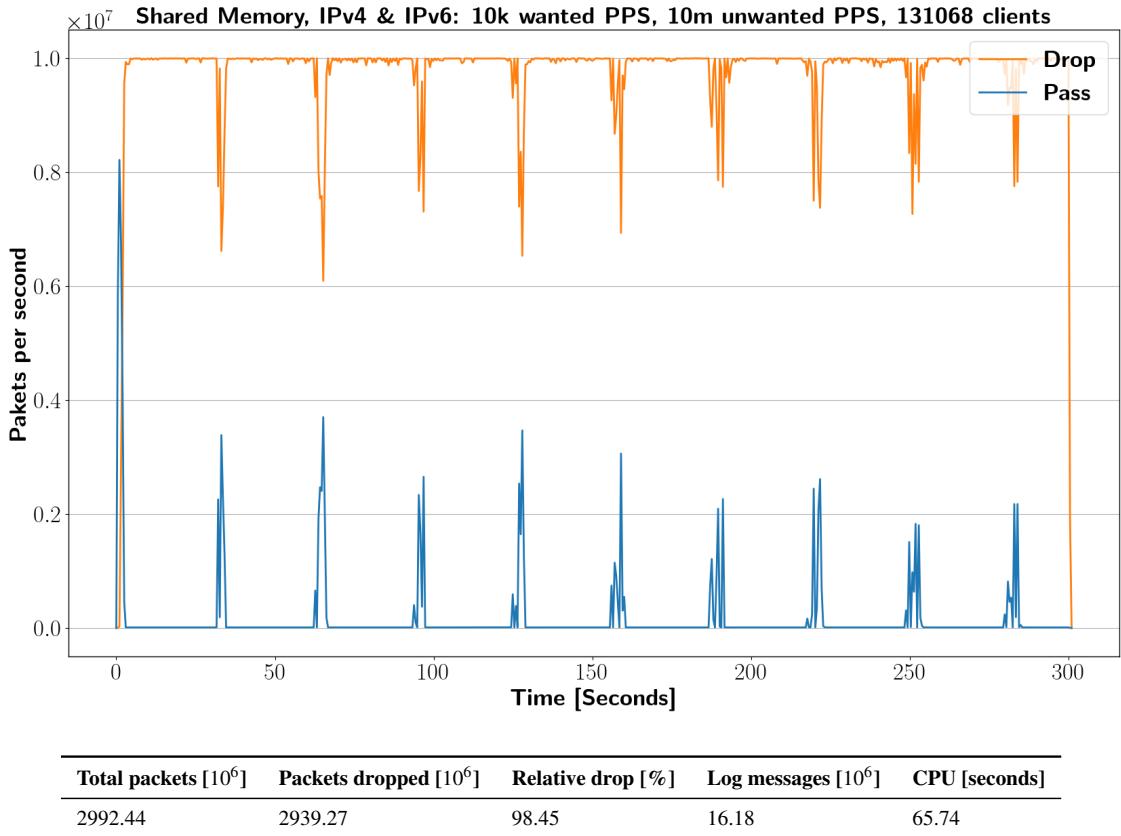


Figure 4.8: Some text

Finally, figure 4.9 presents the results for a supplementary measurement of the shared memory based Simplefail2ban. For this measurements, the invalid traffic rate was increased to 30 million PPS in order to test the boundaries of the system in terms of traffic. Overall, performance was surprisingly good, since over 99% of invalid traffic was successfully dropped. However, the results are likely skewed, as the system was not able handle the entire incoming traffic. As evident by the variance in the drop curve, without corresponding increases in the number of passes packets, not all packets where parsed by the eBPF program. In total, only around 8.095 billion packets out of 9.003 billion packets were registered. Similar behavior was already observed by Florian Mikolajczak [1] for traffic rates beyond 30 million packets per second.

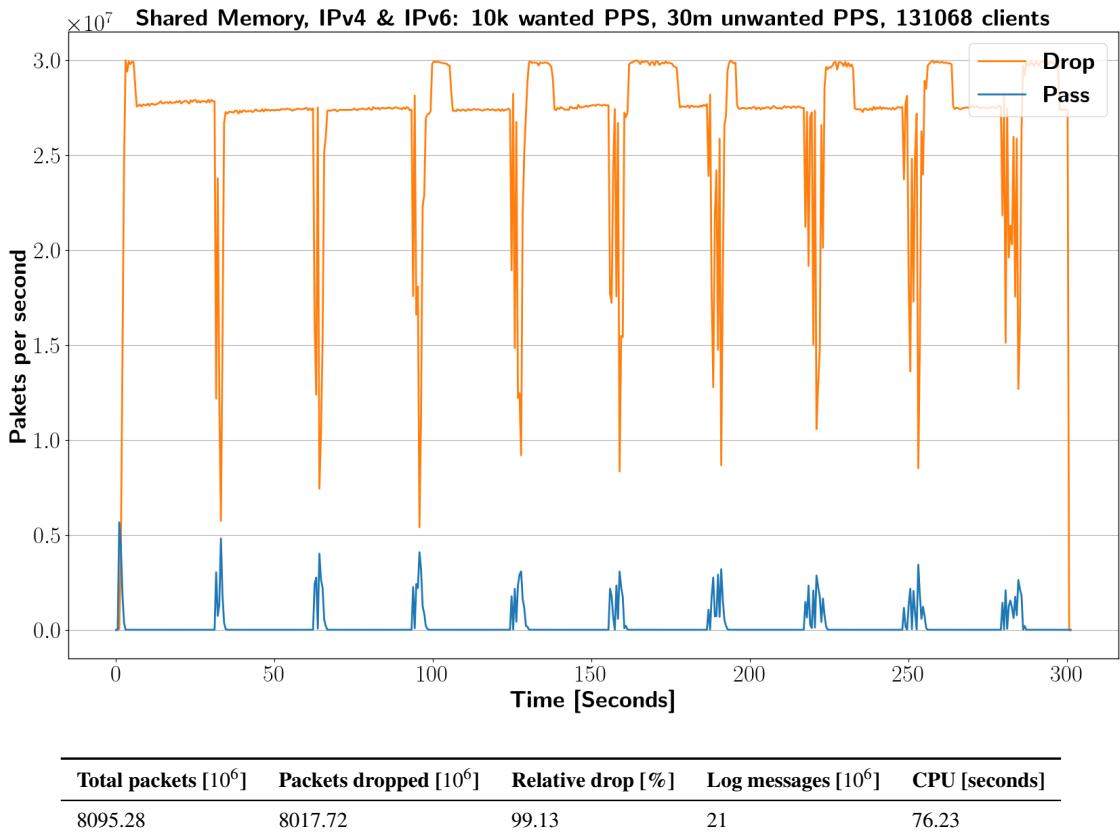


Figure 4.9: Some text

4.4.3 Experiment 4: Shared Memory Features

In the following, features of the shared memory implementation are evaluated. All measurements are conducted with same traffic and client levels as the measurements presented in figure ???. Hence, the measurement servers as a baseline for comparing performance difference with additionally enabled features.

Figure 4.10 presents the results for shared memory Simplefail2ban with an additional reader attached to the buffer. For this purpose, the simplelogstash program introduced in ?? was attached as a second reader, writing the contents of the buffer to a logfile. Compared to the baseline measurement, performance drops by about one percentage point to 97.4% of dropped invalid traffic. Additionally, the variance of when client are unbanned is noticeably larger than in most of the previous measurements. This may caused by the fact, that the test server has to drop messages, as buffer reaches capacity due to the slower second reader. About 54% of all log messages by the server are dropped.

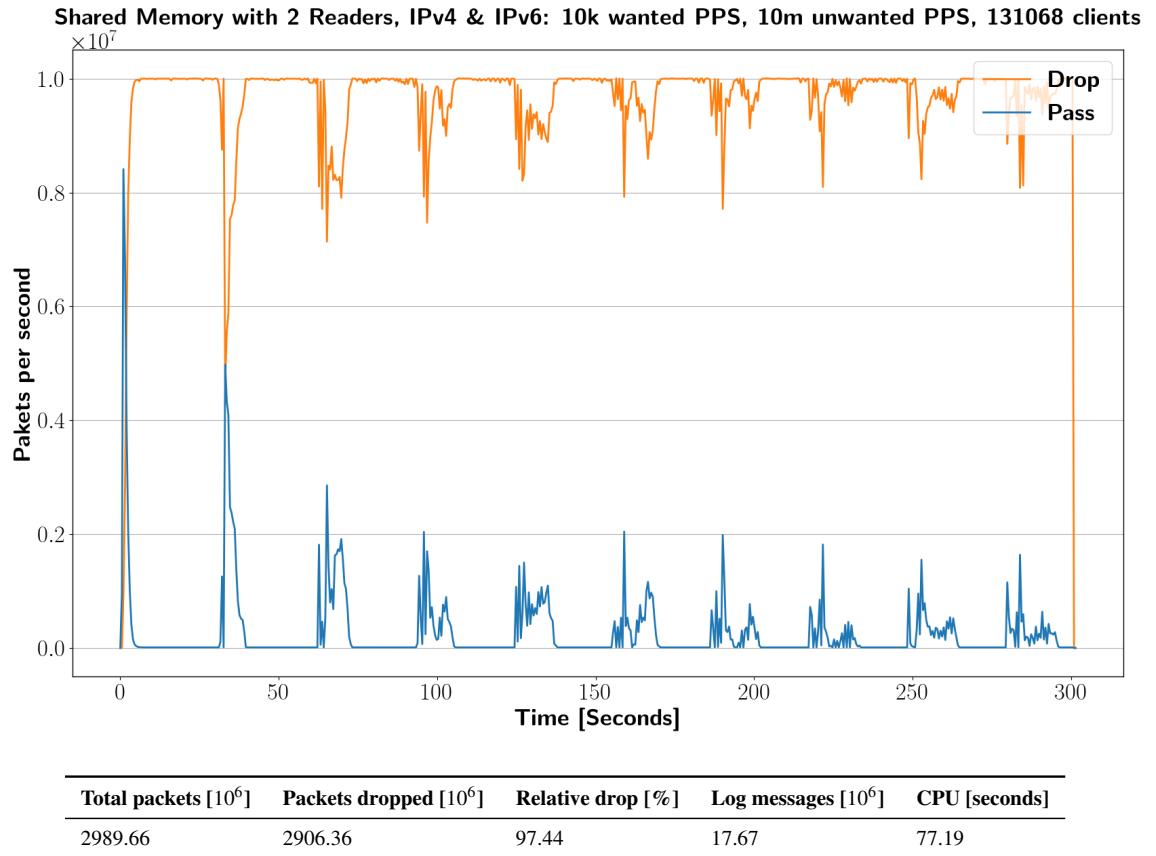


Figure 4.10: Some text

Figure 4.10 presents the results for the two reader scenario with the additionally enabled overwrite feature. Contrary to expectation, the performance is not significantly better, at less than a percentage point of dropped invalid traffic. Simplefail2ban only managed to read about 58% of all log messages written by the server, while the second reader read only about 41% of all messages. This behavior is unexpected and points at a potential bug in the implementation of the overwrite feature. For potential further development of Simplefail2ban, the overwrite feature should be inspected closely and potentially revised, if performance does not improve.

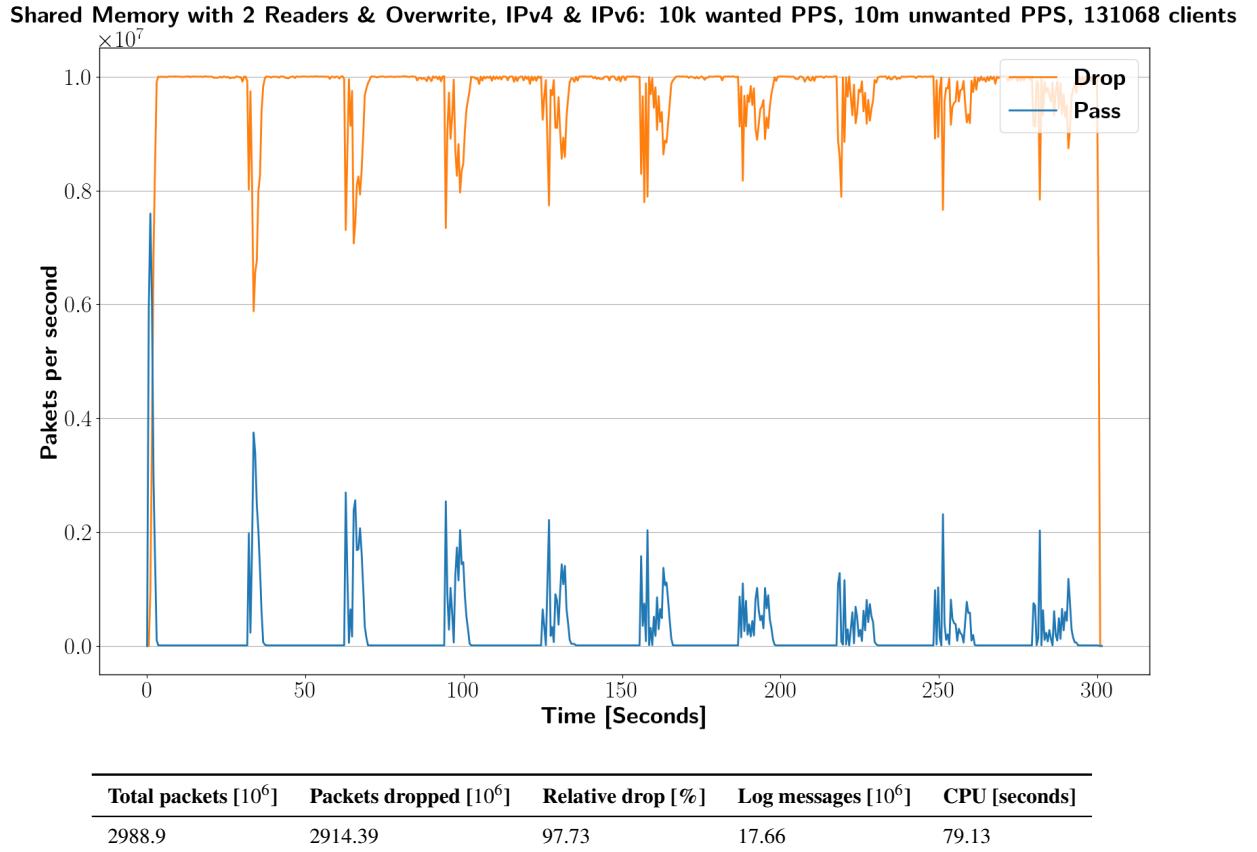


Figure 4.11: Some text

Figure 4.12 presents the results for shared memory Simplefail2ban with the additionally enabled workload stealing feature. Performance does not differ significantly from the baseline measurement. However, the test scenario is also not ideal for testing workload stealing, as the number of log messages was relatively equal distributed across the buffer segment. Still, about 12% of all processed log messages were read via workload stealing. Workload sharing would be expected to have larger performance impact, when distribution of messages is highly unequal, as stealing should become more frequent. Creating a specialized test for this scenario could be an objective for future development.

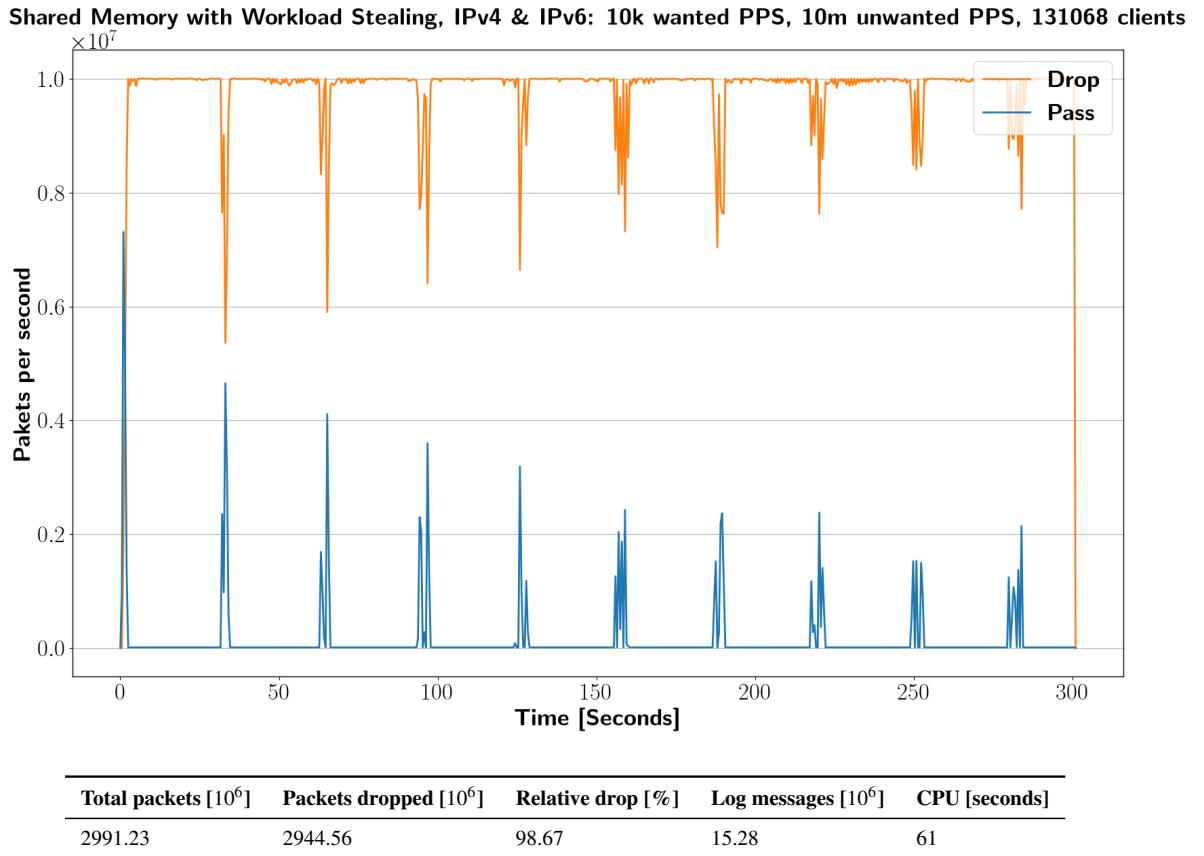


Figure 4.12: Some text

Finally, figure 4.13 display the result for shared memory Simplefail2ban with directly logged IP addresses, instead of Regex matching. This measurement is conducted, to observe the performance impact of the Regex matching on the overall performance of Simplefail2ban. The overall performance does not differ from the baseline measurement. Interestingly, the CPU utilization drops by 50% compared to the baseline. This potentially indicates, that the performance of Simplefail2ban is bottlenecked by the speed of the write operations to the eBPF maps. Measuring the this performance could be a further objective for future development.

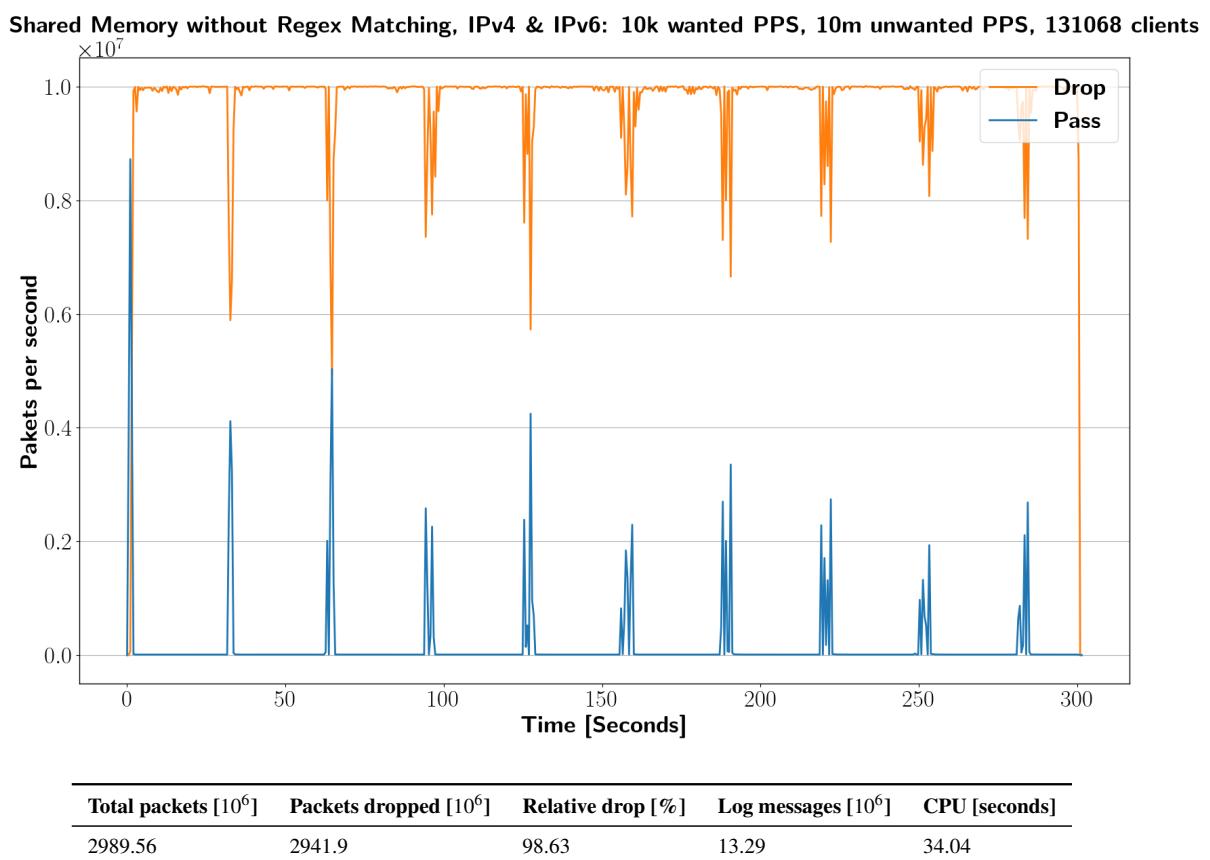


Figure 4.13: Some text

5 Conclusion

List of Figures

2.1	Fail2ban measurement by [1]	4
3.1	IPC Architecture	10
3.2	Shared Memory Architecture	12
3.3	Simplefail2ban Architecture	21
4.1	Fail2ban Replication: 1m PPS, 254 Clients	30
4.2	Fail2Ban Replication: 10k PPS, 1016 Clients	31
4.3	Simplefail2ban, Logfile IPv4 & IPv6, 100k PPS	33
4.4	Simplefail2ban, Logfile IPv4 & IPv6, 1m PPS	34
4.5	Simplefail2ban, Logfile IPv4 & IPv6, 10m PPS	35
4.6	Simplefail2ban, Shared Memory, IPv4 & IPv6, 100k PPS	36
4.7	Simplefail2ban, Shared Memory, IPv4 & IPv6, 1m PPS	37
4.8	Simplefail2ban, Shared Memory, IPv4 & IPv6, 10m PPS	38
4.9	Simplefail2ban, Shared Memory, IPv4 & IPv6, 30m PPS	39
4.10	Simplefail2ban, Shared Memory 2 Readers	40
4.11	Simplefail2ban, Shared Memory 2 Readers with Overwrite	41
4.12	Simplefail2ban, Shared Memory with Workload Sharing	42
4.13	Simplefail2ban, Shared Memory without Regex Matching	43
C.1	Simplefail2ban, Logfile IPv4, 100k PPS	50
C.2	Simplefail2ban, Logfile IPv4, 1m PPS	51
C.3	Simplefail2ban, Logfile IPv4, 10m PPS	52
C.4	Simplefail2ban, Logfile IPv6, 100k PPS	53
C.5	Simplefail2ban, Logfile IPv6, 1m PPS	54
C.6	Simplefail2ban, Logfile IPv6, 10m PPS	55
C.7	Simplefail2ban, Shared Memory, IPv4, 1m PPS	56
C.8	Simplefail2ban, Shared Memory, IPv4, 1m PPS	57
C.9	Simplefail2ban, Shared Memory, IPv4, 10m PPS	58
C.10	Simplefail2ban, Shared Memory, IPv6, 100k PPS	59
C.11	Simplefail2ban, Shared Memory, IPv6, 1m PPS	60
C.12	Simplefail2ban, Shared Memory, IPv6, 10m PPS	61

List of Tables

3.1	Hash Collisions	23
3.2	IP String Conversion	24
4.1	Testbed Summary	26

List of Algorithms

3.1	Shared Memory Ringbuffer: Initialization and Cleanup	14
3.2	Shared Memory Ringbuffer: Writer Parameters	15
3.3	Shared Memory Ringbuffer: Global Header	15
3.4	Shared Memory Ringbuffer: Writer Segment Header	16
3.5	Shared Memory Ringbuffer: Reader Parameters	16
3.6	Shared Memory Ringbuffer: Reader Segment Header	16
3.7	Shared Memory Ringbuffer: Write API	17
3.8	Shared Memory Ringbuffer: Read API	18
3.9	Asynchronous Getline Function	20
3.10	IP Hash Table	23

A Abbreviations

AMQP	Advanced Message Queue Protocol
CPU	Central Processing Unit
DUT	Device under Test
DoS	Denial of Service
eBPF	extended Berkeley Packet Filter
FIFO	First in First out
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System
IP	Internet Protocol
IPC	Inter-Process Communication
IPS	Intrusion Prevention System
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
IO	Input / Output
NIC	Network Interface Card
NIDS	Network-based Intrusion Detection System
OS	Operating System
PoC	Proof of Concept
PPS	Packets per Second
Regex	Regular Expression
SIEM	Security Information and Event Management
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XDP	eXpress Data Path

B Source Files

For the sake of not having to chop down a forest to print this thesis, no full source files will be appended. The source code is available in a git repository at: <https://gitup.uni-potsdam.de/raatschen/bachelorarbeit>. Access can be requested through me, or the second supervisor Max Schrötter.

C Measurements

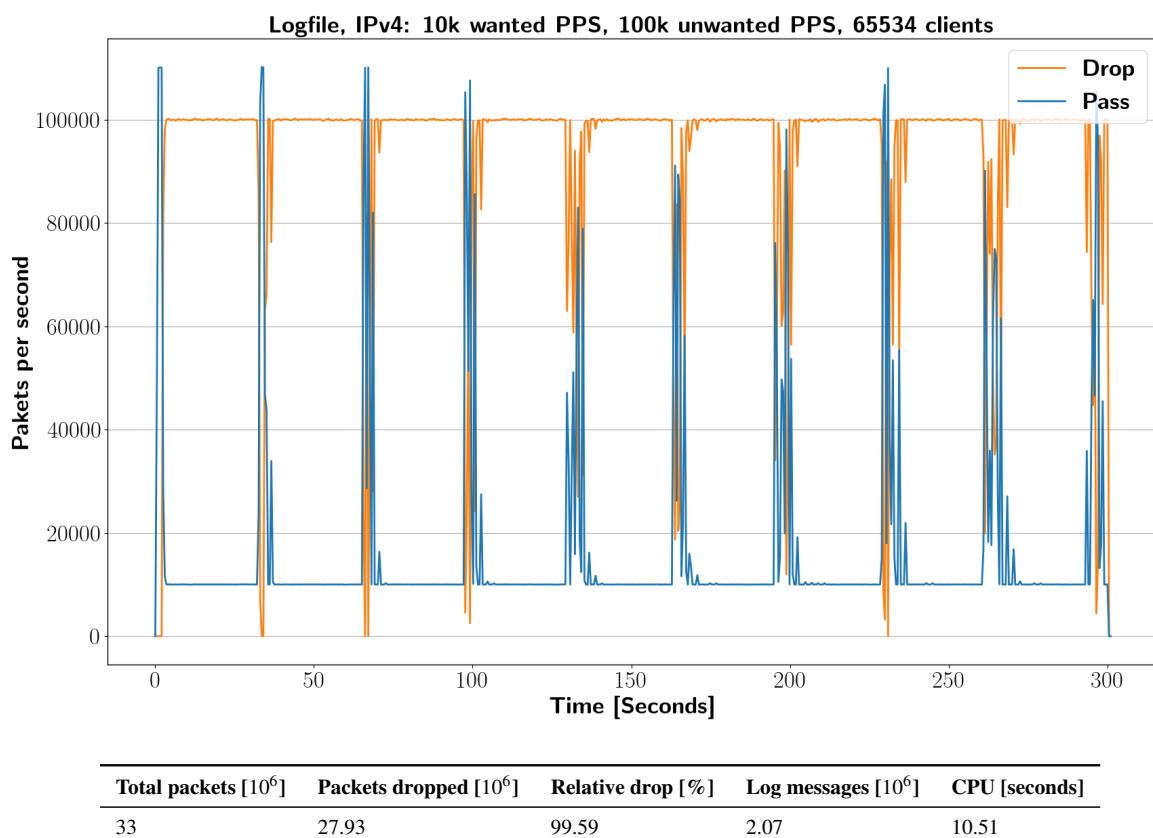


Figure C.1: Some text

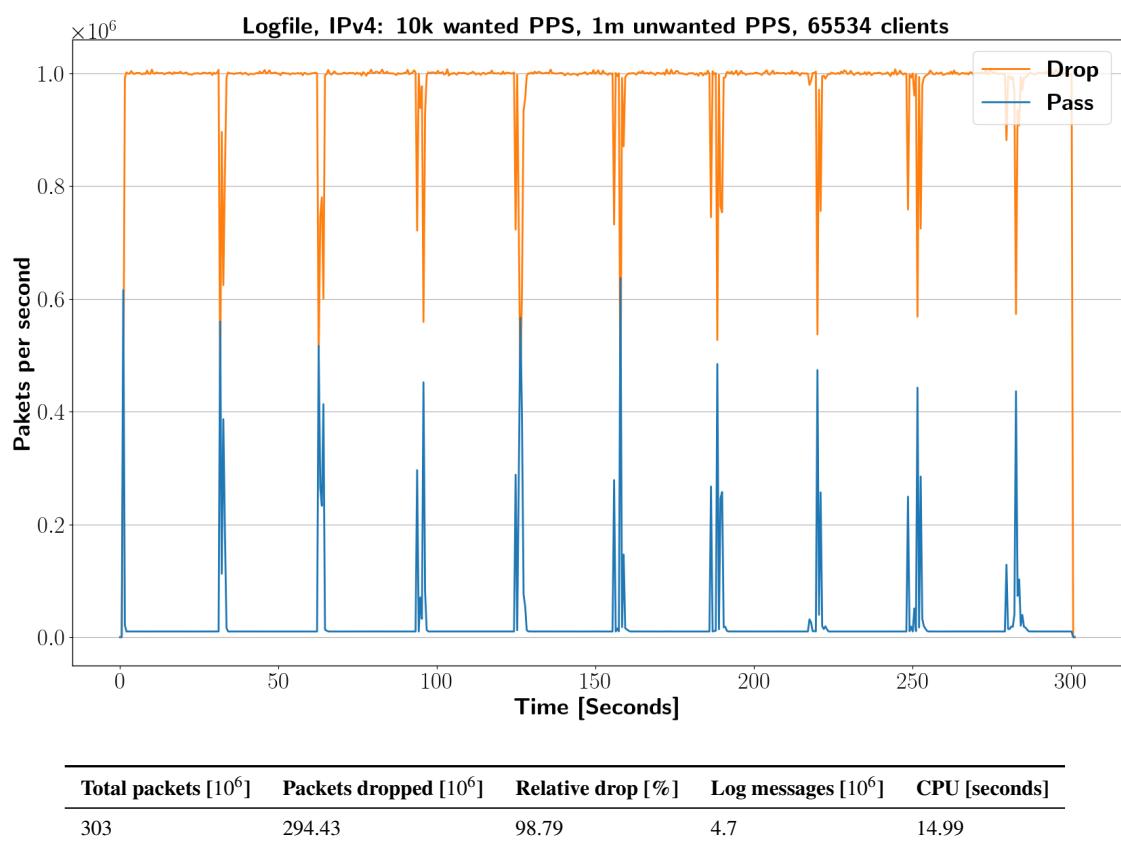


Figure C.2: Some text

C Measurements

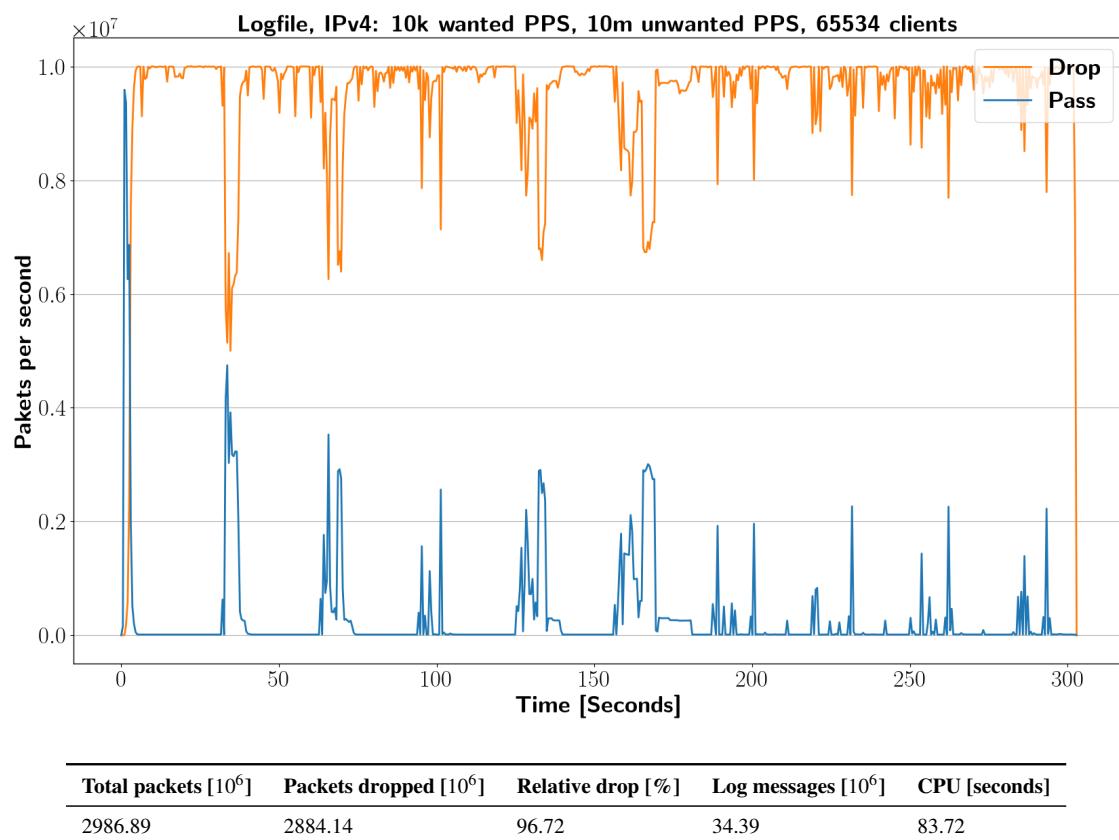


Figure C.3: Some text

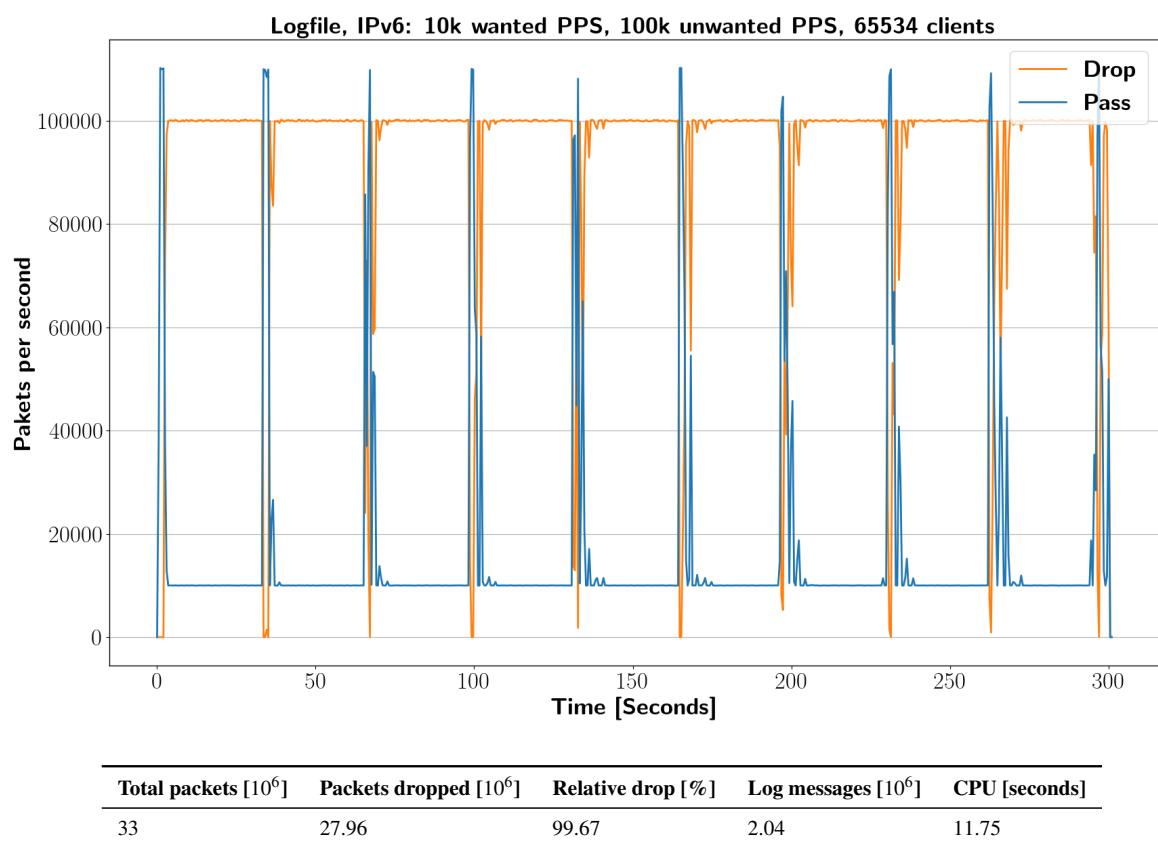


Figure C.4: Some text

C Measurements

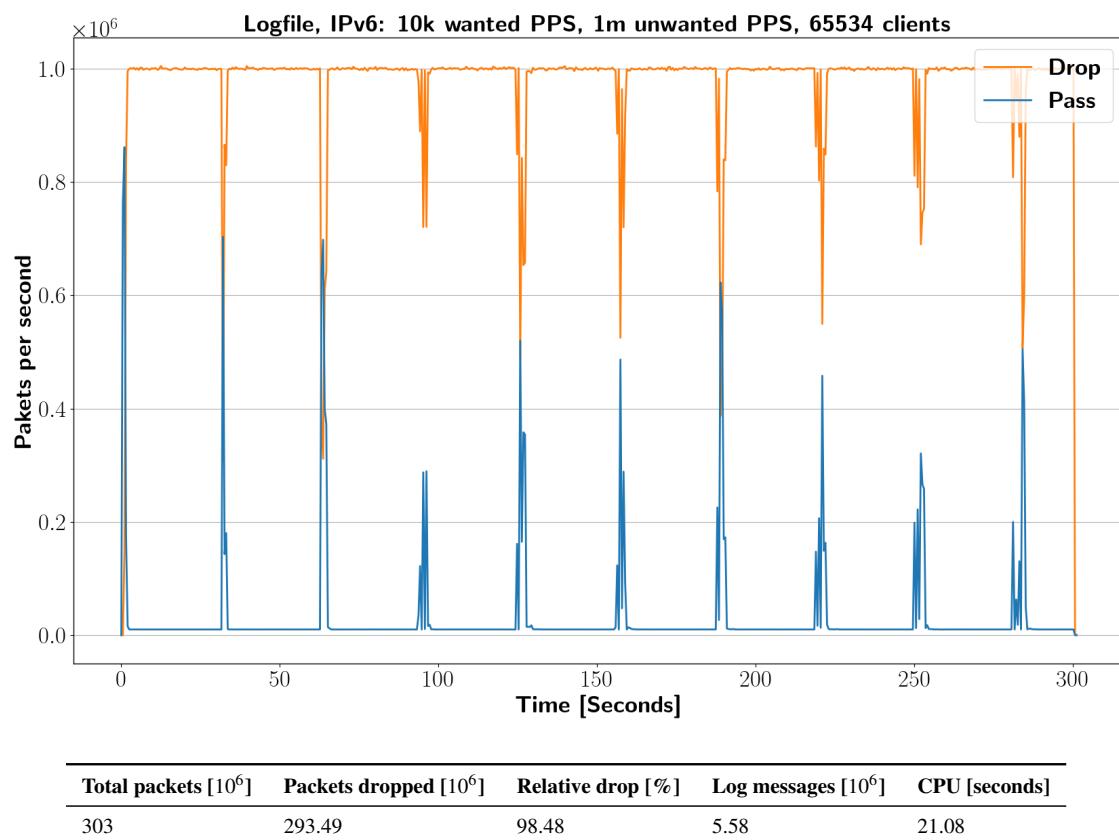


Figure C.5: Some text

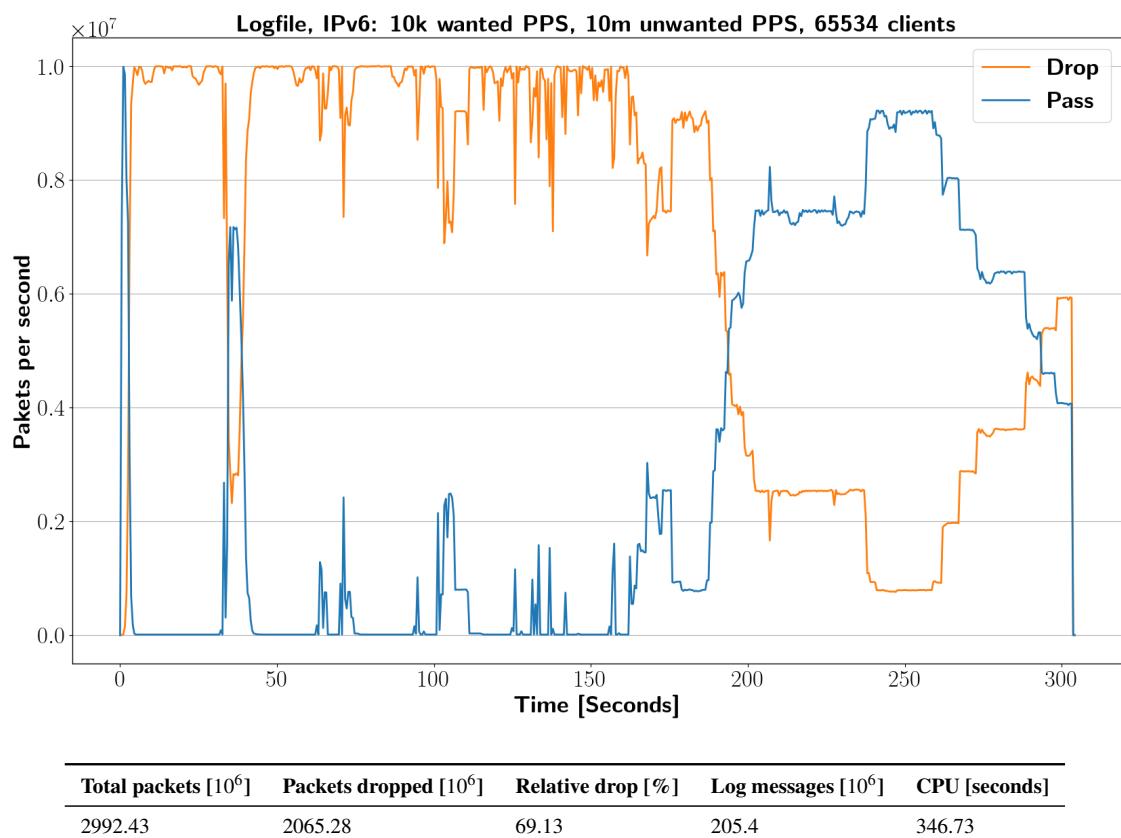


Figure C.6: Some text

C Measurements

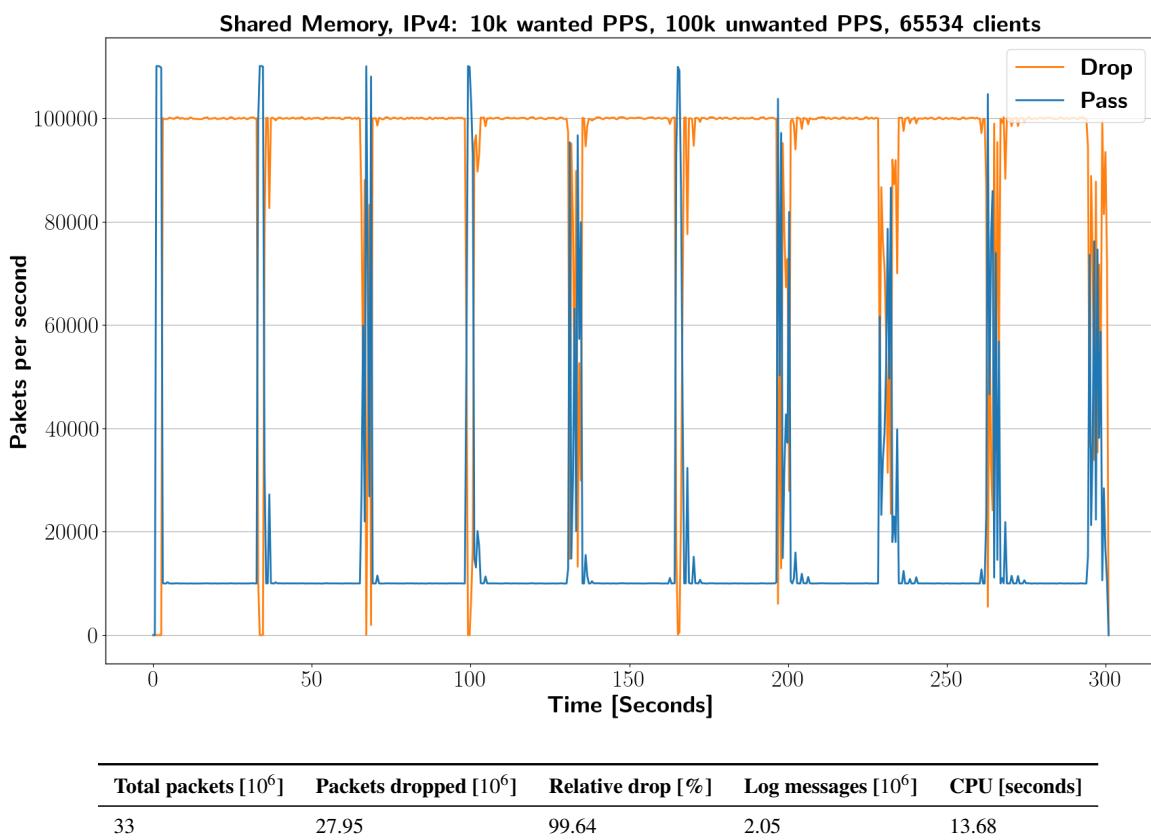


Figure C.7: Some text

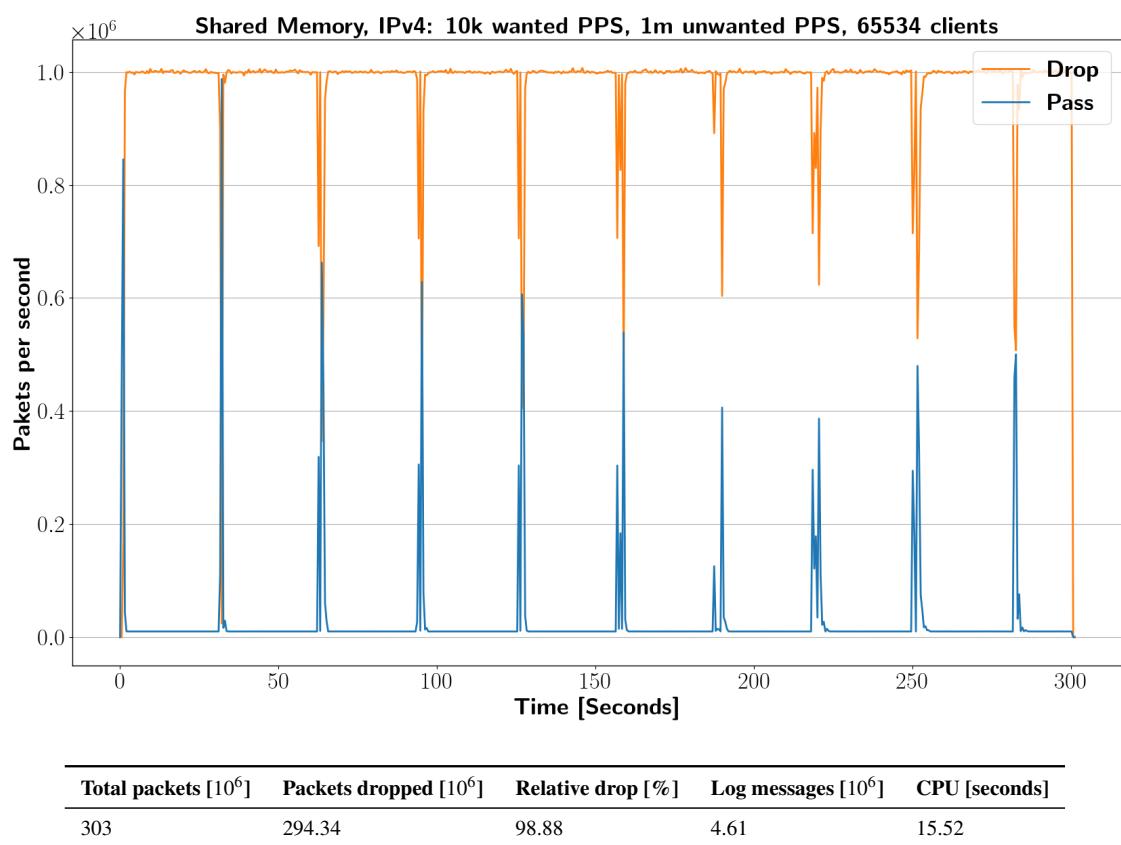


Figure C.8: Some text

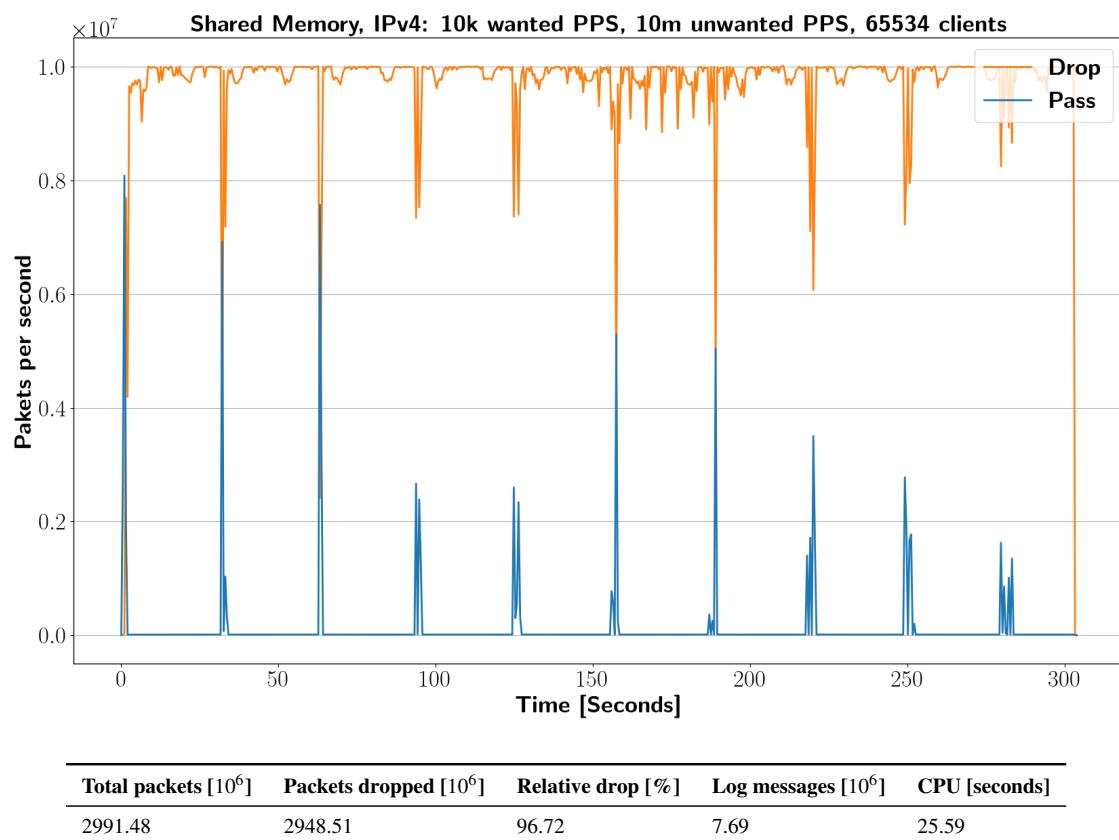


Figure C.9: Some text

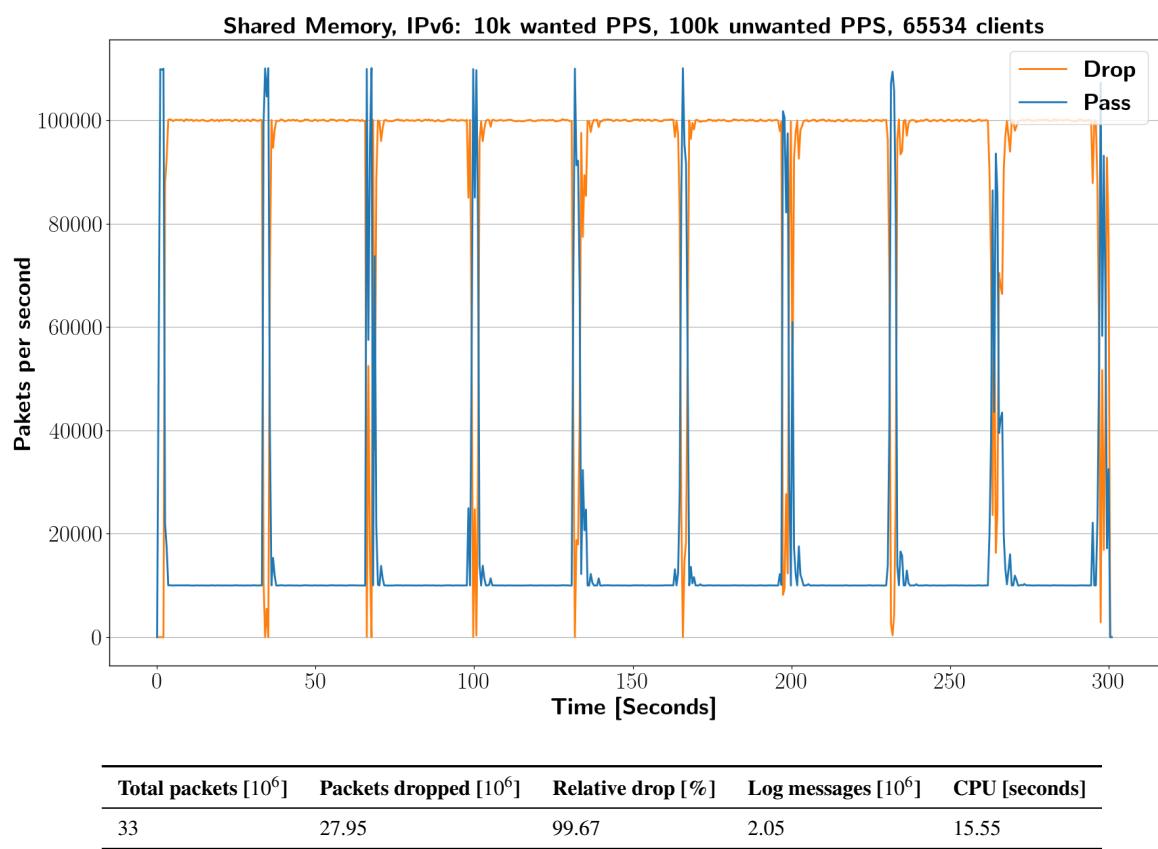


Figure C.10: Some text

C Measurements

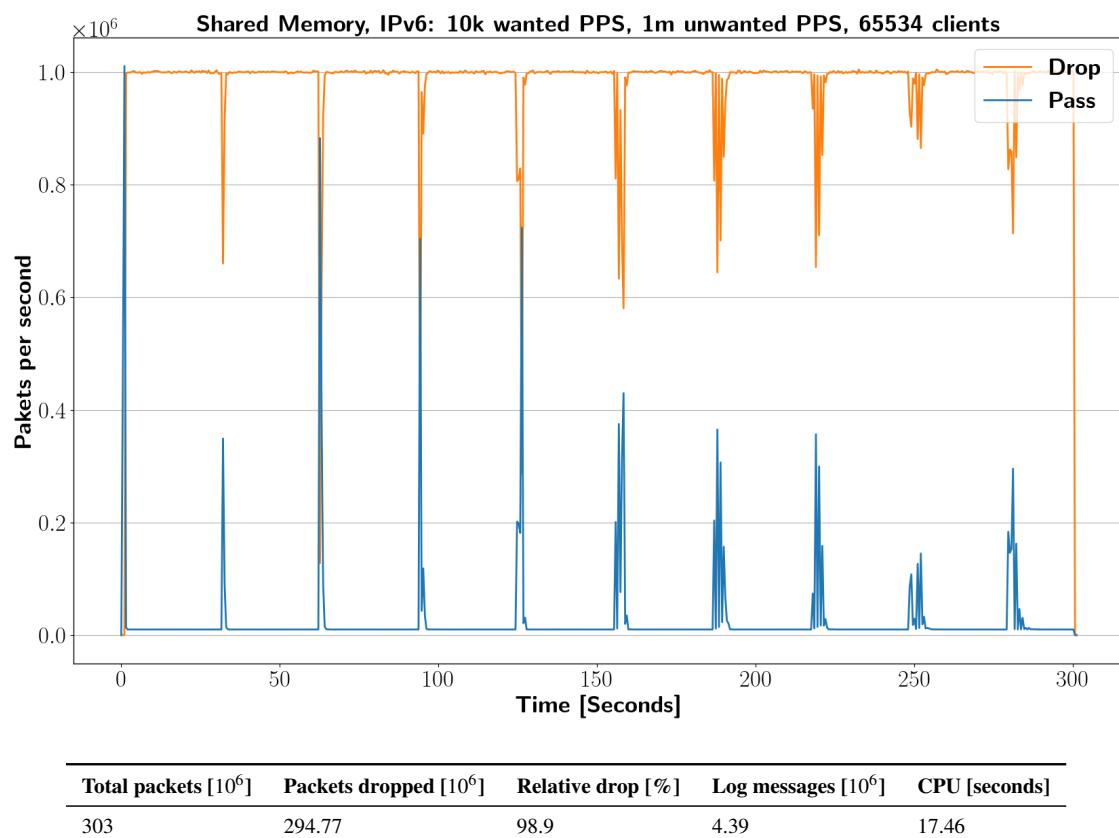


Figure C.11: Some text

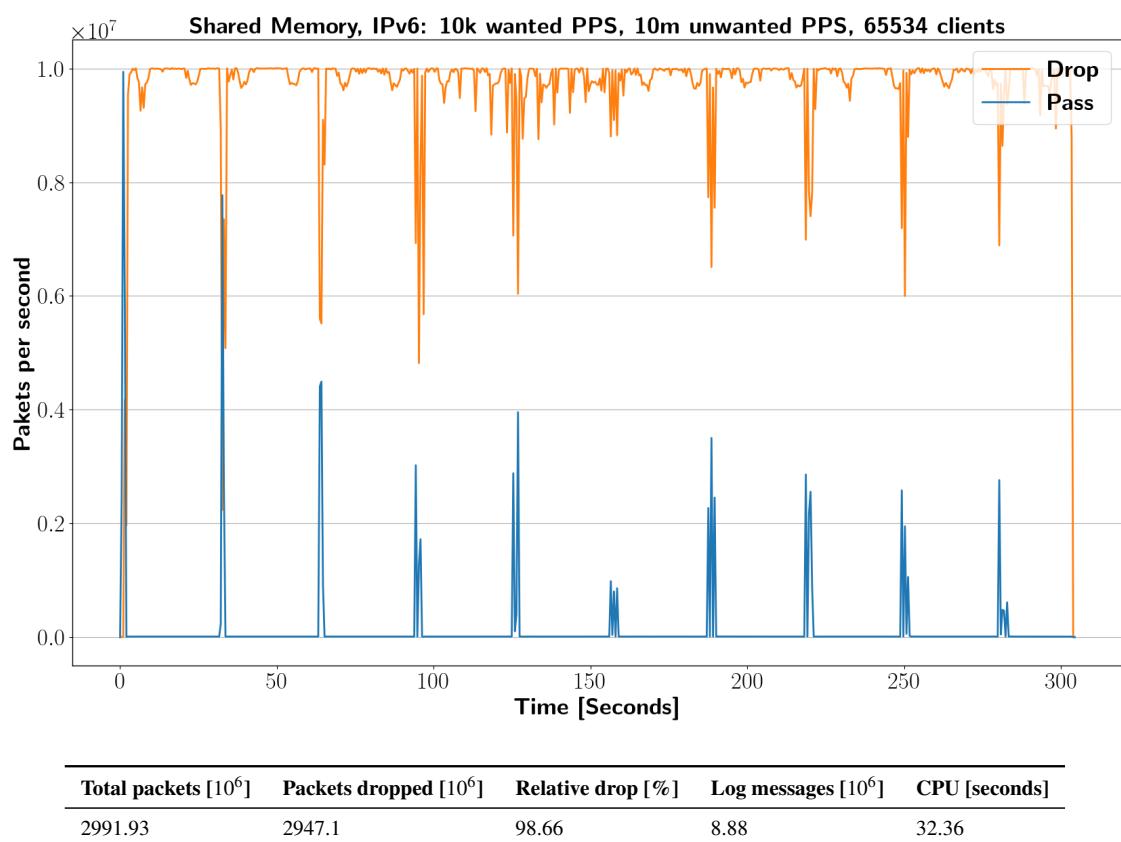


Figure C.12: Some text

Bibliography

- [1] Florian Mikolajczak. Implementation and Evaluation of an Intrusion Prevention System Leveraging eBPF on the Basis of Fail2Ban. <https://www.cs.uni-potsdam.de/bs/teaching/docs/thesis/2022/mikolajczak.pdf>, 2022. Master Thesis, Institute for Computer Science Operating Systems and Distributed Systems, University of Potsdam.
- [2] Giovanni Vigna and Christopher Kruegel. Host-based intrusion detection. In *Handbook of Information Security*, pages 701–713. California State University, 2006.
- [3] James P Anderson. Computer security threat monitoring and surveillance. *Technical Report, James P. Anderson Company*, 1980.
- [4] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, pages 222–232, 1987.
- [5] Fail2Ban. Official Fail2Ban Website. <https://www.fail2ban.org>, 2011. Last visited on: 13.04.2023.
- [6] Netfilter Project. Official netfilter website. <https://www.netfilter.org/>. Last visited on: 11.04.2023.
- [7] Linux man-pages project. Iptables Manpage. <https://www.man7.org/linux/man-pages/man8/iptables.8.html>, 2020. Last visited on: 13.04.2023.
- [8] W Richard Stevens. *UNIX network programming, Volume 2, Interprocess Communications*. Prentice Hall, NJ, USA, 1998.
- [9] Linux man-pages project. Systen V IPC Manpage. <https://man7.org/linux/man-pages/man7/sysvipc.7.html>, 2020. Last visited on: 13.04.2023.
- [10] Linux man-pages project. Posix Shared Memory Manpage. https://man7.org/linux/man-pages/man7/shm_overview.7.html, 2021. Last visited on: 13.04.2023.
- [11] R Recio, B Metzler, P Culley, J Hilland, and D Garcia. Rfc 5040: A remote direct memory access protocol specification, 2007.
- [12] Linux man-pages project. Pipe Manpage. <https://www.man7.org/linux/man-pages/man7/pipe.7.html>, 2021. Last visited on: 12.04.2023.
- [13] W Richard Stevens. *UNIX network programming, Volume 2, Network Programming*. Prentice Hall, NJ, USA, 1998.
- [14] Linux man-pages project. Unix Socket Manpage. <https://man7.org/linux/man-pages/man7/unix.7.html>, 2021. Last visited on: 12.04.2023.

- [15] Linux man-pages project. POSIX Message Queue Manpage. https://man7.org/linux/man-pages/man7/mq_overview.7.html, 2020. Last visited on: 13.04.2023.
- [16] ZeroMQ. Official ZeroMQ Website. <https://zeromq.org/>, 2022. Last visited on: 13.04.2023.
- [17] Rainer Gerhards. Rfc 5424: The syslog protocol, 2009.
- [18] Rainer Gerhards. Rsyslog. <https://www.man7.org/linux/man-pages/man7/pipe.7.html>. Last visited on: 13.04.2023.
- [19] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.
- [20] Elasticsearch B.V. Logstash. <https://www.elastic.co/logstash/>. Last visited on: 13.04.2023.
- [21] Sandeep Bhatt, Pratyusa K Manadhata, and Loai Zomlot. The operational role of security information and event management systems. *IEEE security & Privacy*, 12(5):35–41, 2014.
- [22] Jaein Jeong. High performance logging system for embedded unix and gnu/linux applications. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 310–319. IEEE, 2013.
- [23] Intel. Hyperscan, high-performance regular expression matching library. <https://github.com/intel/hyperscan>. Last visited on: 12.04.2023.
- [24] Jens Axboe. Efficient io with io_uring. https://kernel.dk/io_uring.pdf, 2019. Last visited on: 12.04.2023.
- [25] Jens Axboe. Liburing. <https://github.com/axboe/liburing>. library for the io_uring Linux kernel interface.
- [26] Cisco Systems. Official TRex Website. <https://trex-tgn.cisco.com>. Last visited on: 13.04.2023.
- [27] Scapy. Official Scapy Website. <https://scapy.net/>. Last visited on: 13.04.2023.
- [28] Libzmq. Github issue: Support for so_reuseport. SupportforSO_REUSEPORT, 2015. Last visited on: 13.04.2023.
- [29] Andrew Barrington, Steven Feldman, and Damian Dechev. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1321–1328, 2015.
- [30] Paul Raatschen. Thesis git repository. <https://gitup.uni-potsdam.de/raatschen/bachelorarbeit>, 2023.
- [31] Linux man-pages project. Perf Manpage. <https://www.man7.org/linux/man-pages/man1/perf.1.html>, 2022. Last visited on: 11.04.2023.

Bibliography

- [32] Linux man-pages project. Getline Manpage. <https://www.man7.org/linux/man-pages/man3/getline.3.html>, 2021. Last visited on: 13.04.2023.
- [33] Andi Kleen. Spooky-c. <https://github.com/andikleen/spooky-c>. C version of Bob Jenkins' spooky hash.
- [34] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.